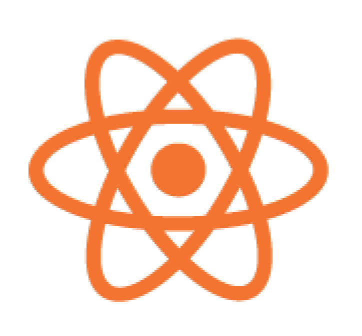


JavaScript ES6
控件应用
布局技术
网络技术
导航栈技术
项目实战

React Native 全教程

移动端跨平台应用开发

• 张益珲 编著 •

实战React Native

清华大学出版社



React Native 全教程

移动端跨平台应用开发

• 张益珲 编著 •

清華大學出版社
北 京

内 容 简 介

本书由经验丰富的移动开发工程师精心编撰，全书从逻辑上可分为 5 个部分，循序渐进地向读者展示使用 React Native 开发跨平台移动应用的全流程，第 1 部分介绍 React Native 语言基础 JavaScript；第 2 部分介绍大量使用于 React Native 开发中的 ECMAScript 6 的新特性；第 3 部分介绍 React Native 开发技巧，包括独立组件应用、布局技术、网络技术、导航栈技术等；第 4 部分通过 3 个实战项目手把手地教读者开发完整的 React Native 应用；第 5 部分介绍 React Native 的一些高级技巧，比如和原生交互、嵌入原生应用、React Native 组件开发等。本书特别对 React Native 在开发 iOS 和 Android 跨平台应用时给出范例效果对比演示，现场感十足。

本书既适合想快速上手 React Native 的初学者、有 Android 和 iOS 开发基础想构建跨平台移动应用的开发人员使用，也可用作培训机构和大中专院校的教学参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

React Native 全教程：移动端跨平台应用开发/张益珲编著. —北京：清华大学出版社，2018
ISBN 978-7-302-49813-1

Ⅰ. ①R… Ⅱ. ①张… Ⅲ. ①移动终端－应用程序－程序设计－教材 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2018）第 037615 号

责任编辑： 王金柱
封面设计： 王　翔
责任校对： 闫秀华
责任印制： 王静怡

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者： 北京泽宇印刷有限公司
经　　销： 全国新华书店
开　　本： 190mm×260mm　　**印　　张：** 25.5　　**字　　数：** 653 千字
版　　次： 2018 年 4 月第 1 版　　**印　　次：** 2018 年 4 月第 1 次印刷
印　　数： 1~3000
定　　价： 79.00 元

产品编号：075000-01

前　　言

首先，笔者十分荣幸也十分高兴你选择本书来学习 React Native 的开发。由于笔者的个人能力有限，这本书可能并不是最完美最优秀的，但是相信无论你的基础如何，都可以随着本书的学习与练习，完完整整地独立开发出自己的 React Native 应用，因为笔者也是这么学过来的。因此，将本书定义为教程，不如称其为一本经验与总结的笔记，相信你在学习的过程中会深有体会。

说起来，JavaScript 的语法并不是这本书的核心，但是学习一种功夫之前，一定要有一把顺手的武器，因为依然有很多初学者或原生开发者对 JavaScript 语言的了解并不深入，所以本书的前 4 章着重对 JavaScript 语法以及 ES6 的新特性进行介绍，帮助你为后边的学习扫除基础障碍。

学习客户端编程，最重要的莫属界面、数据、逻辑这 3 部分，本书的第 5～8 章将向你介绍 React Native 中的基础界面组件、数据与网络技术以及用户交互管理技术等，学习一门技能就是在完成一张大拼图，每一个知识点都是这张拼图中的一块，学习完这 4 章内容，你将掌握 React Native 开发中所有的基础技能，后面就是对它们的组合和应用了。

本书第 9～11 章提供了 3 个完整的 React Native 实战练习，这 3 章的项目也将由简到难，帮助你熟练应用前面所学习的知识。

本书第 12 章为扩展章节，这一章节将更偏向介绍 React Native 的一些高级开发技术，比如和原生交互、嵌入原生应用、开发 React Native 组件等，如果你有兴趣，可以好好研究一下。

IT 领域日新月异，React Native 是一种移动端跨平台软件开发框架，可能并不是最优秀的，但是 Facebook 的长期维护和它优秀的设计思想无论如何都是值得我们学习的。有人说，每学习一门技术，每次离开自己所擅长的领域走向新的领域学习都是一种重生，这个过程可以让你感受到不同的思维模式，体验到不同圈子的乐趣。和你一样，笔者也是一名学习者，如果你愿意，可以随时和笔者交流，QQ：316045346。

为方便读者上机练习，本书提供了全书实例源代码，下载地址：

https://pan.baidu.com/s/1msOpjsdGcoSRCN5K4qcTPQ（注意区分数字和英文字母大小写）

如果你在下载过程中遇到问题，可发邮件至 booksaga@126.com，邮件标题为“React Native 全教程：移动端跨平台应用开发下载资源”，获得帮助。

最后，再次感谢你选择了本书，笔者也真心地希望它可以帮助你到达自己的预定目标。这本书最终能呈现在你的面前，除了笔者的努力，还要感谢支持我的家人和朋友，尤其是王金柱编辑，在写作过程中他给了我巨大的帮助与鼓励。

珲　少

2018 年 2 月

目　　录

第 1 章　从 JavaScript 开始 …………………… 1

1.1　学习环境的配置 ……………………………… 1

1.1.1　使用浏览器进行 JavaScript 代码的调试 …………………………………… 1

1.1.2　使用 Sublime Text 工具来编写 JavaScript 代码 ………………………… 3

1.1.3　安装 Sublime Text 插件管理器 PackageControl …………… 3

1.1.4　使用 PackageControl 进行 JavaScript 代码智能提示插件的安装 ………… 5

1.1.5　安装 JavaScript 代码格式化插件 ……………………………………… 7

1.1.6　在 Sublime Text 中运行 JavaScript 代码 ……………………………………… 7

1.2　初识 JavaScript ……………………………… 8

1.2.1　JavaScript 的语法特点 ……………… 9

1.2.2　JavaScript 中的变量 ………………… 10

1.3　JavaScript 中的数据类型 …………… 12

1.3.1　原始类型 …………………………………… 13

1.3.2　引用类型 …………………………………… 16

1.4　JavaScript 中的运算符 ………………… 18

1.4.1　算术运算符 ………………………………… 18

1.4.2　赋值运算符 ………………………………… 21

1.4.3　关系运算符 ………………………………… 22

1.4.4　逻辑运算符 ………………………………… 24

1.4.5　位运算符 …………………………………… 26

1.4.6　特殊运算符 ………………………………… 30

1.4.7　运算符的优先级与结合性 ……… 32

第 2 章　JavaScript 流程控制与函数 …… 34

2.1　条件分支结构 ……………………………… 34

2.1.1　if-else 分支结构 ……………………… 34

2.1.2　switch-case 分支结构 ………………35

2.2　循环结构 ……………………………………37

2.2.1　while 循环结构 ………………………37

2.2.2　for 循环结构 ……………………………38

2.3　中断与跳转结构 …………………………39

2.3.1　break 语句 ………………………………39

2.3.2　continue 语句 …………………………41

2.4　异常捕获结构 ……………………………42

2.4.1　使用 throw 语句抛出异常 ………43

2.4.2　异常的捕获与处理 ……………………44

2.4.3　异常的传递 ………………………………46

2.5　JavaScript 中的函数 ……………………48

2.5.1　使用函数语句定义函数 ……………48

2.5.2　使用函数表达式定义函数 ………50

2.5.3　使用 Function 构造函数 …………51

第 3 章　JavaScript 对象基础 ……………… 52

3.1　初识 JavaScript 对象 …………………52

3.1.1　在 JavaScript 中创建对象 ………52

3.1.2　设置对象的属性和行为 ……………54

3.2　JavaScript 中常用的内置对象 ………55

3.2.1　JavaScript 中的 Number 对象 ……55

3.2.2　JavaScript 中的 String 对象 ………57

3.2.3　JavaScript 中的 Boolean 对象 ……59

3.2.4　JavaScript 中的 Array 对象 ………60

3.2.5　JavaScript 中的 Date 对象 …………64

3.2.6　JavaScript 中的 Math 对象 …………67

3.2.7　JavaScript 中的 RegExp 对象 ……69

3.2.8　JavaScript 中的 Function 对象 ……73

3.3　深入 JavaScript 中的 Object 对象 …75

3.3.1　为对象属性进行配置 …………………76

3.3.2 Object 构造方法对象中的常用函数 ······ 77
3.3.3 Object 实例对象中的常用方法 ······ 82
3.4 面向对象编程技术 ······ 82
3.4.1 JavaScript 中模拟类的方式 ······ 83
3.4.2 在 JavaScript 中实现继承机制 ··· 86

第 4 章 ECMAScript 6 新特性 ······ 91

4.1 ECMAScript 6 的块级作用域 ······ 91
4.1.1 let 关键字 ······ 92
4.1.2 const 关键字 ······ 94
4.2 解构赋值 ······ 95
4.2.1 数组的解构赋值 ······ 95
4.2.2 对象的解构赋值 ······ 96
4.2.3 字符串与函数参数的解构赋值 ······ 98
4.3 箭头函数 ······ 99
4.3.1 箭头函数的基本用法 ······ 99
4.3.2 箭头函数中 this 的固化 ······ 100
4.4 Set 与 Map 数据结构 ······ 102
4.4.1 Set 集合结构 ······ 102
4.4.2 Map 字典结构 ······ 104
4.5 Proxy 代理 ······ 106
4.5.1 使用 Proxy 代理对对象的属性读写进行拦截 ······ 106
4.5.2 Proxy 代理处理器支持的拦截操作 ······ 108
4.6 Promise 承诺对象 ······ 110
4.6.1 Promise 对象执行异步任务 ······ 110
4.6.2 Promise 任务链 ······ 112
4.6.3 Promise 对象组合 ······ 113
4.7 Generator 生成器与 yield 语句 ······ 115
4.7.1 Generator 函数应用 ······ 115
4.7.2 Generator 任务参数的传递 ······ 117
4.8 使用 class 定义类 ······ 119
4.8.1 使用 class 定义类 ······ 119
4.8.2 class 类的继承 ······ 120
4.9 模块引入 ······ 121
4.9.1 export 关键字 ······ 121
4.9.2 import 关键字 ······ 122
4.9.3 默认导出与导入 ······ 122

第 5 章 React Native 开发环境的搭建 ······ 124

5.1 iOS 开发环境的搭建 ······ 124
5.1.1 申请 AppleID 账号 ······ 124
5.1.2 安装 Xcode 开发工具 ······ 125
5.2 Android 开发环境的搭建 ······ 126
5.2.1 下载 Android Studio 开发工具 ······ 126
5.2.2 安装相关 SDK 和模拟器 ······ 127
5.3 React Native 开发环境配置 ······ 130
5.3.1 安装 React Native 构建环境 ······ 130
5.3.2 运行你的第一个 React Native 应用 ······ 131

第 6 章 React Native 独立组件基础篇 ······ 134

6.1 Text 文本组件的应用 ······ 134
6.1.1 文字风格设置 ······ 134
6.1.2 Text 组件属性的设置 ······ 138
6.1.3 Text 组件的嵌套 ······ 140
6.1.4 React Native 程序的调试 ······ 141
6.2 Button 按钮组件的应用 ······ 142
6.2.1 Button 组件的简单使用 ······ 142
6.2.2 小应用：屏幕霓虹灯 ······ 144
6.3 Image 图像组件的应用 ······ 145
6.3.1 渲染图像的方式 ······ 145
6.3.2 Image 组件的风格自定义 ······ 148

6.3.3 Image 组件的属性和方法解析 ········ 151
6.4 Switch 开关组件的应用 ········ 154
6.5 Slider 滑块组件的应用 ········ 156
6.6 ActivityIndicator 指示器组件的应用 ········ 159
6.7 TextInput 用户输入组件的应用 ···· 160
6.8 StatusBar 状态栏组件的应用 ········ 165
6.9 Picker 选择器组件的应用 ········ 167
6.10 Modal 模态视图组件的应用 ······ 169
6.11 KeyboardAvoidingView 组件的应用 ········ 171
6.12 WebView 网页组件的应用 ········ 174
6.12.1 WebView 常用属性解析 ········ 174
6.12.2 WebView 加载过程监听相关属性 ········ 177
6.12.3 React Native 与 WebView 交互 ········ 178
6.13 View 视图组件的应用 ········ 179
6.13.1 View 组件 Style 属性的解析 ········ 180
6.13.2 View 组件基础属性的解析 ········ 182
6.14 Touchable 相关交互组件的应用 ········ 183
6.14.1 TouchableWithoutFeedback ····· 184
6.14.2 TouchableOpacity ········ 185
6.14.3 TouchableNativeFeedback ······ 186
6.14.4 TouchableHighlight ········ 188
6.15 ScrollView 滚动视图组件的应用 ········ 189
6.15.1 ScrollView 的基础用法 ········ 189
6.15.2 ScrollView 常用属性解析 ······ 190
6.15.3 手动设置 ScrollView 组件的滚动位置 ········ 192
6.16 ListView 列表组件的应用 ········ 193
6.16.1 使用 DataSource 渲染 ListView 视图 ········ 193
6.16.2 ListView 属性方法解析 ········ 197
6.17 高性能列表组件 FlatList ········ 199
6.17.1 创建一个简单的 FlatList 列表视图 ········ 199
6.17.2 FlatList 中常用方法解析 ········ 202
6.18 分区列表组件 SectionList 的应用 ········ 202
6.19 RefreshControl 刷新组件的应用 ········ 205
第 7 章 React Native 独立组件高级篇 ········ 208
7.1 时间选择器 DatePickerIOS 组件的应用 ········ 208
7.2 DrawerLayoutAndroid 抽屉组件的应用 ········ 209
7.3 进度条组件的应用 ········ 211
7.3.1 通过文件名分平台加载组件 ····· 212
7.3.2 ProgressBarAndroid 组件常用属性 ········ 213
7.3.3 ProgressViewIOS 组件常用属性 ········ 214
7.4 SegmentedControlIOS 组件的应用 ········ 214
7.5 Android 平台上的工具条组件 ······ 215
7.6 Navigator 导航控制器 ········ 218
7.6.1 Navigatior 牛刀小试 ········ 219
7.6.2 Navigator 属性配置 ········ 220
7.6.3 Navigator 实例方法解析 ········ 221
7.7 iOS 平台的导航控制器 NavigatorIOS 组件 ········ 222
7.7.1 使用 NavigatorIOS 组件 ········ 222

7.7.2 NavigatorIOS 属性与方法解析 ··· 225
7.8 标签栏 TabBarIOS 组件 ··· 226

第 8 章 React Native 技能进阶 ··· 230

8.1 React Native 布局技术 ··· 230
8.1.1 布局中的主轴与次轴 ··· 231
8.1.2 精准定义组件的尺寸 ··· 234
8.1.3 相对定位与绝对定位 ··· 237
8.2 React Native 中的颜色定义 ··· 240
8.3 警告弹窗的应用 ··· 242
8.3.1 Alert 组件的应用 ··· 243
8.3.2 iOS 平台专用警告框 AlertIOS ··· 245
8.4 ActionSheetIOS 抽屉视图的应用 ··· 247
8.4.1 普通功能列表抽屉 ··· 247
8.4.2 分享视图抽屉 ··· 248
8.5 自定义组件的属性与使用样式表 ··· 250
8.5.1 自定义组件的属性 ··· 250
8.5.2 通过 StyleSheet 样式表定义组件的风格 ··· 251
8.6 Android 平台的时间选择器 ··· 252
8.7 Android 平台悬浮提示信息 Toast 的应用 ··· 254
8.8 监听与控制 Android 设备返回键的行为 ··· 255
8.9 监听程序运行状态 ··· 257
8.10 跨平台的分享功能 ··· 258
8.11 监听键盘事件 ··· 260
8.12 React Native 网络技术 ··· 262
8.12.1 使用 fetch 方法进行网络请求 ··· 262
8.12.2 使用 XMLHttpRequest 进行网络请求 ··· 264
8.13 进行用户位置获取 ··· 266
8.14 数据持久化技术 ··· 267
8.15 剪贴板工具的应用 ··· 270
8.16 获取设备网络状态 ··· 271
8.17 React Native 动画技术 ··· 273
8.17.1 创建单值驱动的动画 ··· 273
8.17.2 使用 timing 方法执行平滑过渡动画 ··· 275
8.17.3 深入理解 easing ··· 276
8.17.4 二维动画对象与衰减动画 ··· 278
8.17.5 弹簧动画 ··· 280
8.17.6 Interpolation 插值动画 ··· 281
8.17.7 聚合动画值 ··· 282
8.17.8 组合动画 ··· 283
8.17.9 循环动画 ··· 285
8.17.10 布局动画 ··· 286
8.17.11 自定义组件动画 ··· 287
8.18 调用设备振动模块 ··· 288
8.19 封装滑动手势 ··· 289
8.20 获取屏幕尺寸信息 ··· 292
8.21 特定平台代码 ··· 293
8.22 定时器的简单应用 ··· 294

第 9 章 实战项目：汇率转换器 ··· 296

9.1 搭建汇率转换器项目主界面 ··· 297
9.2 显示屏面板的初步开发 ··· 299
9.3 货币类型切换功能开发 ··· 302
9.4 键盘界面设计 ··· 306
9.5 实现汇率转换器核心功能 ··· 310

第 10 章 实战项目：微信热门精选 ··· 315

10.1 申请免费的 API 服务 ··· 315
10.2 搭建项目网络模块 ··· 317
10.3 搭建文章列表界面 ··· 319

10.4　文章目录视图与首页导航栏完善……322
10.5　文章详情页面的开发……326
10.6　为文章列表页添加下拉刷新与上拉加载更多功能……329

第 11 章　实战项目：掌上新闻……332

11.1　应用结构搭建……332
11.2　完善标题栏组件……335
11.3　进行网络模块的开发……338
11.4　使用列表展示数据……339
11.5　完善新闻目录列表……341
11.6　标题栏与页面联动开发与优化加载逻辑……344
11.7　使用导航进行页面跳转……348
11.8　完善下拉刷新与上拉加载更多功能……351
11.9　完善导航栏……353
11.10　添加收藏夹功能……356
11.11　优化方向与应用图标设置……361

第 12 章　React Native 高级技巧……363

12.1　直接操作组件的属性……363
12.2　对 React Native 版本进行升级……365
12.3　React Native 的更多调试技巧……366
12.4　React Native 插件开发……367
12.4.1　构建 iOS 工程的原生模块……367
12.4.2　构建 Android 工程的原生模块……371
12.4.3　深入了解原生模块的函数参数……373
12.5　封装原生 UI 组件……375
12.5.1　封装 iOS 平台的原生 UI 组件……375
12.5.2　开发 Android 跑马灯组件……382
12.6　在原生工程中嵌入 React Native 模块……387
12.6.1　将 iOS 工程的某个模块进行 React Native 化……387
12.6.2　将 Android 工程的某个模块进行 React Native 化……391
12.7　在真机上运行 React Native 工程……397

第 1 章

从 JavaScript 开始

JavaScript 是一门流行的脚本语言，同时，它也是开发 React Native 跨平台应用的基础语言。本书前几章会安排了较多篇幅来介绍 JavaScript，以帮助读者零门槛地进行 React Native 开发。当然，如果你是熟练的 Web 开发者，相信 JavaScript 对你来说并不能成为障碍，但是依然建议你阅读第 4 章的内容——ES6 的一些新特性，因为 ES 在 React Native 框架中随处可见。

本章，将简单地对 JavaScript 作全面的了解，同时配置好开发环境，以便于开启你的学习之旅。

1.1 学习环境的配置

编程是一种手脑结合的工作，我们在学习编程时，动手远比读书重要得多。因此在 JavaScript 语言学习正式开始之前，首先需要搭建完善、方便、易用的开发环境。有了称手的工具，在学习的路上，才能披荆斩棘，一往无前。

1.1.1 使用浏览器进行 JavaScript 代码的调试

JavaScript 语言最初设计时用于作为浏览器的脚本语言，因此所有浏览器都可以对 JavaScript 代码进行解释与运行。MacOS 的 Safari 浏览器、Google 的 Chrome 浏览器都带有强大的开发者工具。以 Chrome 浏览器为例（在百度上搜索 Chrome，可以十分容易地下载到 Chrome 的最新版本），可以借助 HTML 文件来进行 JavaScript 代码的运行，先来编写一段简单的 HTML 代码，如下：

```
<html>
<head>
    <title></title>
```

```
        <script type="text/javascript">
            console.log("Hello World");
        </script>
    </head>
    <body>
    </body>
    </html>
```

用任何文本编辑器来写这段 HTML 代码都没有问题，这里推荐使用 Sublime Text 编辑器——一款十分优秀的跨平台代码编写软件，最后将此 HTML 文件的后缀保存为 html。

使用 Chrome 浏览器打开此 HTML 文件，会发现页面上一片空白，这没什么好奇怪的，这里只是编写了一个空的网页模板，其中嵌入了一句 JavaScript 脚本代码，除此之外，我们什么也没有做，那么如何来验证 JavaScript 代码的执行情况呢？打开 Chrome 浏览器的开发者工具，如图 1-1 所示。

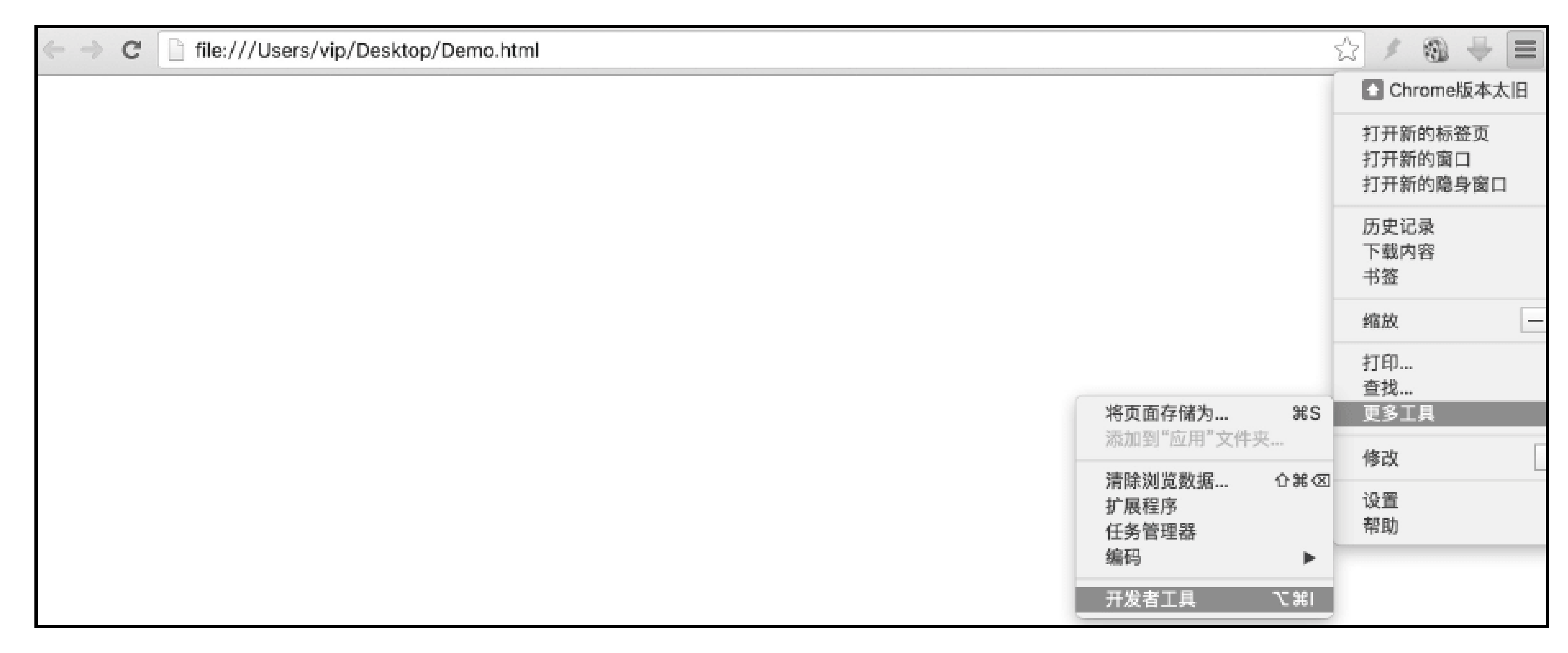


图 1-1　打开 Chrome 浏览器的开发者工具

之后可以看到，界面被分成了 3 部分，如图 1-2 所示。其中，网页区将显示实时的网页效果，功能区可以查看 HTML 标签、所加载的网络数据、加载耗时信息等。打印调试区中可以查看打印信息，也可以进行 JavaScript 代码的调试。如图 1-2 所示，打印区中打印出了我们在 HTML 代码中嵌入的“Hello World”语句，这就是 JavaScript 版的 HelloWorld 工程！



图 1-2　Chrome 浏览器的开发者工具

每次在 HTML 文件中编写一些 JavaScript 测试代码都要重新刷新浏览器，这对开发者来说是一件无法忍受的事情，有时，我们只是想知道某一行代码的执行结果，可以直接在开发者工具的打印调试区进行代码的编写与运行。例如，在其中输入如图 1-3 所示的代码。

```
Console  Emulation  Rendering
<top frame> ▼  Preserve log
> var name = 'jaki';
  var age = '25';
  console.log('name:'+name+'  '+'age:'+age);
```

图 1-3　直接在 Chrome 浏览器中进行 JavaScript 代码的调试

抽 丝 剥 茧

默认情况下，当在 Chrome 开发工具的调试区输入一行代码后，如果敲击 Enter 键会直接运行此行代码，若需要多行输入，需要使用 Shift+Enter 组合键来进行换行。

1.1.2　使用 Sublime Text 工具来编写 JavaScript 代码

Chrome 浏览器提供的开发者工具虽然强大，但是只适用于已完成项目的检查与调试，无法用来进行项目的开发。可以编写 JavaScript 代码的编辑器十分多，例如专门开发大型 Web 项目的 WebStorm 工具、小巧的用于开发移动端网页的 HBuilder 工具、通用编辑器 Sublime Text 工具等。对于 JavaScript 语法部分的学习，强烈建议大家使用 Sublime Text 工具。首先，Sublime Text 十分轻量，占内存极小，运行极快。其次，Sublime Text 有大量的插件支持，能够很好地提供代码高亮、智能补全、代码格式化等高级开发工具所提供的功能。最重要的是，Sublime Text 可以配置编译系统，有了它，我们就不再需要依赖浏览器来进行 JavaScript 代码的运行与调试了。

在如下网站可以下载到 Sublime Text 工具的最新版本：http://www.sublimetext.com/。

下载安装完成 Sublime Text 工具后，其已经自带了代码高亮的功能，可以进行 JavaScript 代码的编写，但是并没自动补全、代码格式化与运行 JavaScript 代码的功能，这些将会在后面一一解决，把 Sublime Text 工具武装成一款强大的 JavaScript 编辑器。

1.1.3　安装 Sublime Text 插件管理器 PackageControl

Sublime Text 工具的插件十分丰富，但是如何快速地找到并安装所需要的插件依然不容易。PackageControl 工具可以帮助我们解决这些问题。

在 Sublime Text 中有两种方式可以进行 PackageControl 插件的安装。

（1）第一种方式是直接在 Sublime Text 的控制台输入如下代码，之后按 Enter 键来进行 PackageControl 工具的安装：

```
    import urllib.request,os; pf = 'Package Control.sublime-package'; ipp =
sublime.installed_packages_path();urllib.request.install_opener( urllib.reques
t.build_opener( urllib.request.ProxyHandler()) ); open(os.path.join(ipp, pf),
'wb').write(urllib.request.urlopen( 'http://sublime.wbond.net/' + pf.replace(' ',
'%20')).read())
```

在 Sublime Text 工具中使用 contorl+`组合键可以直接打开控制台，将上面的代码直接复制进去即可，如图 1-4 所示。

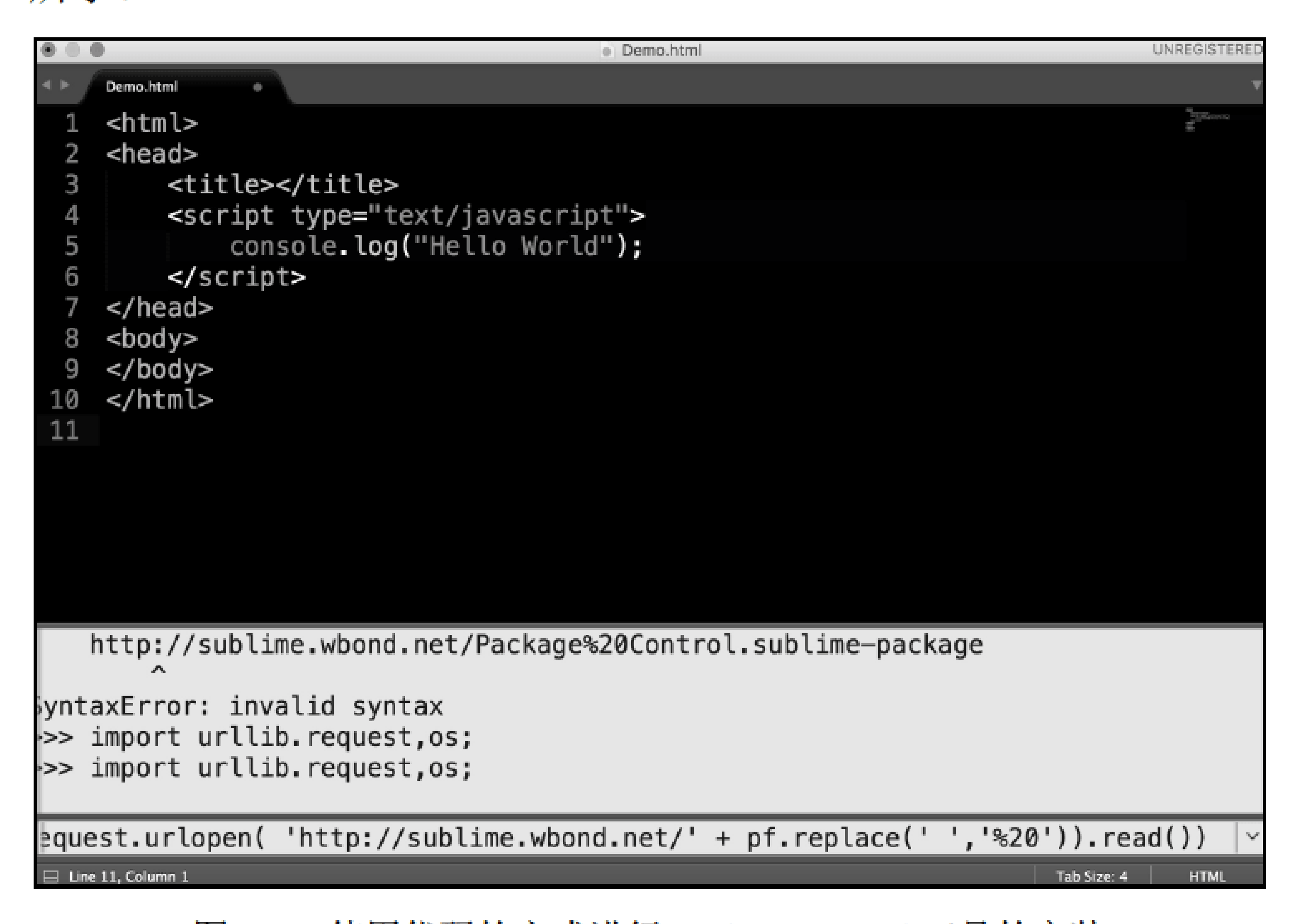


图 1-4　使用代码的方式进行 PackageControl 工具的安装

由于网络环境与 Sublime Text 版本更新有太多的不可控性，使用代码进行 PackageControl 工具的安装不一定会成功，我们也可以直接下载 PackageControl 插件进行安装。可以从如下地址下载 PackageControl 插件：http://sublime.wbond.net/ Package%20Control.sublime-package。

下载完成后，如图 1-5 所示，打开 Sublime Text 工具的插件目录 Preferences→Browse Packages。将下载到的 PackageControl 安装文件放入 Installed Packages 文件夹中，如图 1-6 所示。

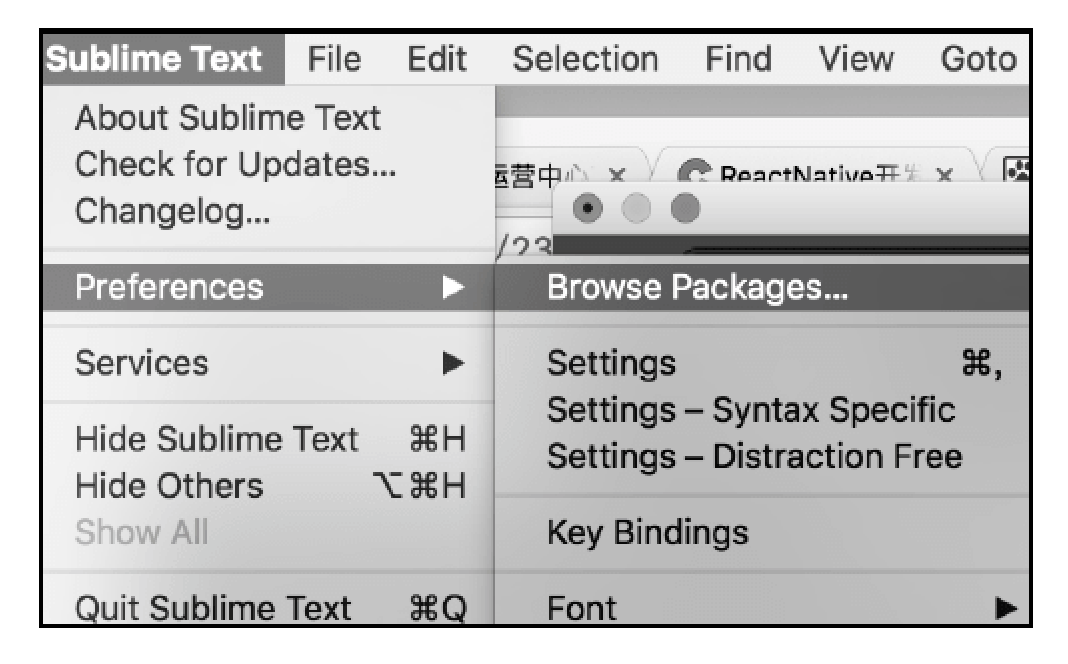



图 1-5　打开 Sublime Text 的插件目录

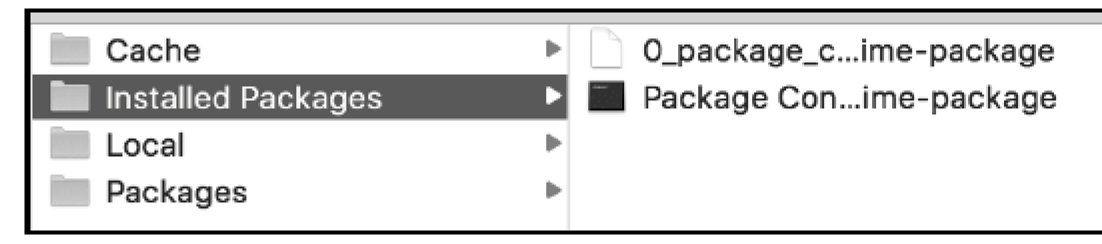


图 1-6　将 PackageControl 安装文件放入 Installed Packages 文件夹中

之后将 Sublime Text 完全关掉，重启 Sublime Text，如果可以在 Preferences 中看到 Package Control 项目，说明安装已经成功，如图 1-7 所示。

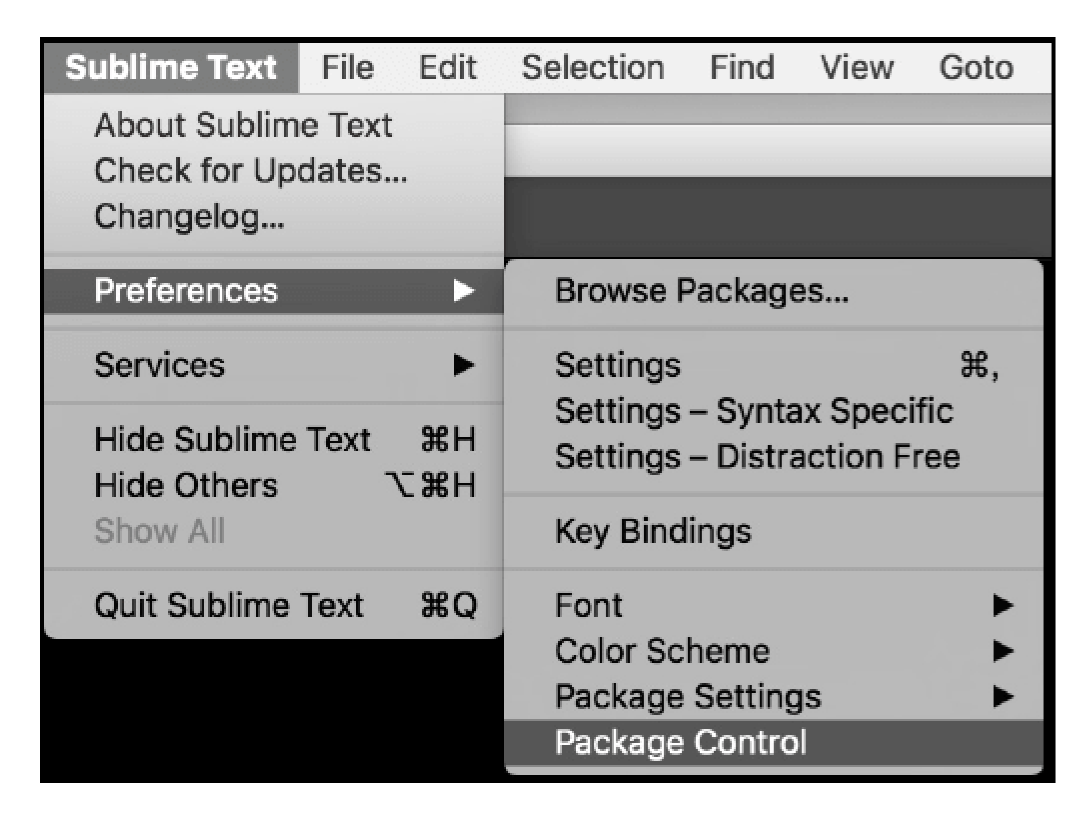


图 1-7 安装成功 PackageControl 工具

抽 丝 剥 茧

上面提供的下载 PackageControl 安装包的地址服务器在国外，国内访问时常会出现波动，如果你无法从上面的地址下载 PackageControl 安装包，可以尝试如下地址：http://zyhshao.github.io/file/Package%20Control.sublime-package。

1.1.4 使用 PackageControl 进行 JavaScript 代码智能提示插件的安装

对代码的智能提示是高级编辑工具必备的一项功能。SublimeCodeIntel 是一个全功能的代码自动提示插件，其支持众多流行语言的代码智能提示，例如 JavaScript、HTML、CSS、Python、PHP 等。

使用 PackageControl 可以十分方便地进行 SublimeCodeIntel 插件的安装，首先打开 Sublime Text 编辑工具，Mac 电脑使用 Command+Shift+P 组合键来打开命令行，在其中输入“package control:”来检索 PackageControl 工具所提供的命令，也可以通过 Sublime Text→Preferences→Package Control 直接打开 PackageControl 命令行，如图 1-8 所示。

PackageControl 工具中提供了许多易用的命令，如安装插件、查看已安装的插件列表、删除插件等，如图 1-9 所示。

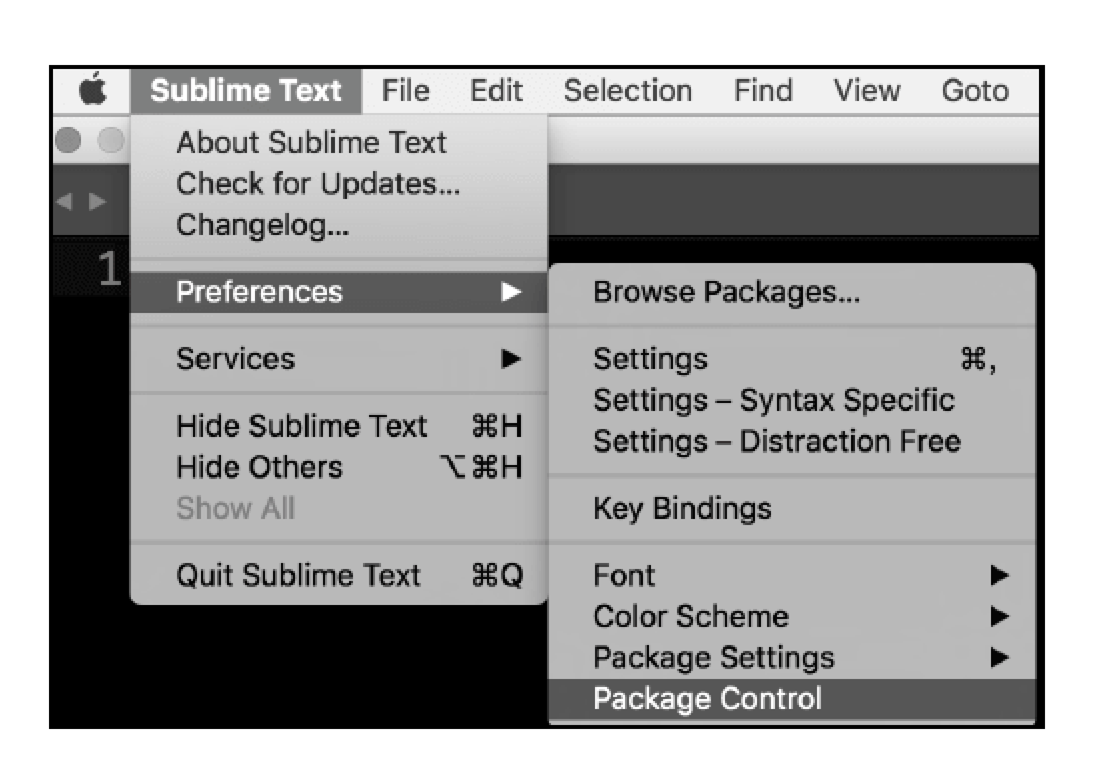


图 1-8 打开 PackageControl 命令行

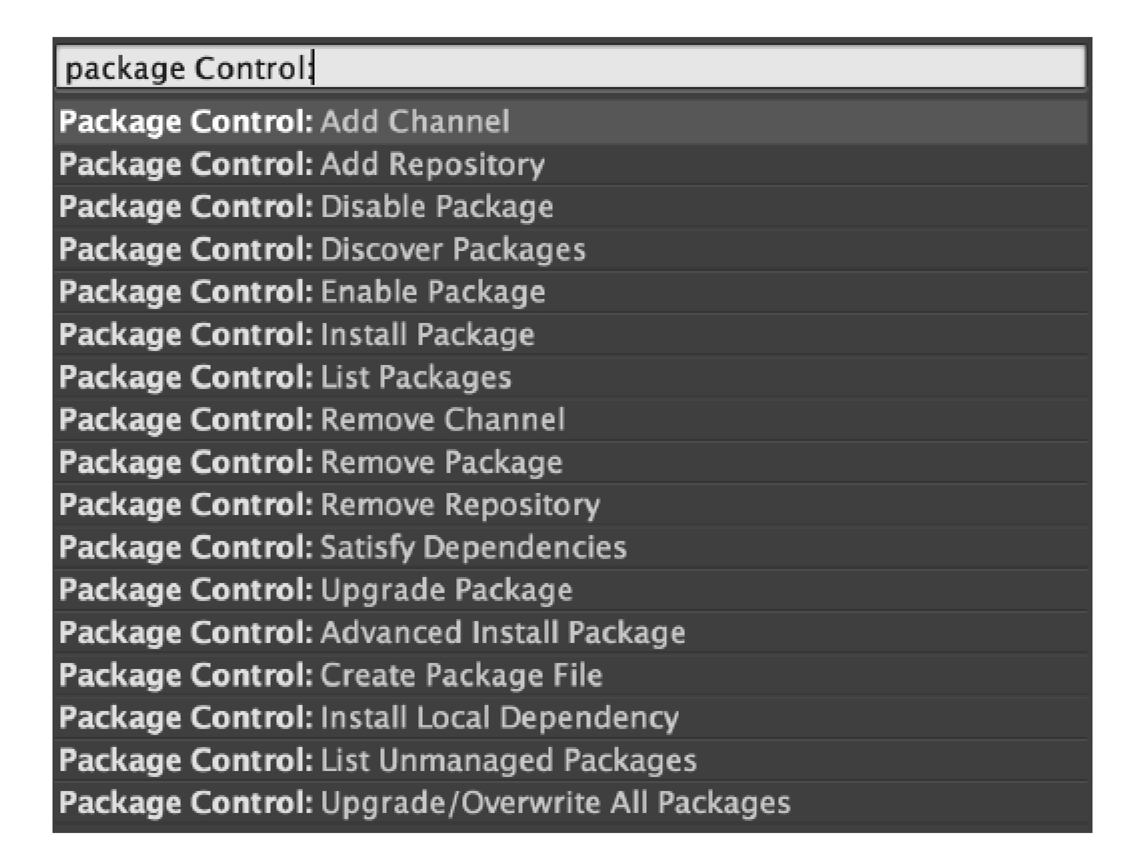


图 1-9 PackageControl 工具提供的命令

输入“Install Package”后按 Enter 键进入插件安装命令。此时会进入插件列表，如图 1-10 所示。

在其中输入“SublimeCodeIntel”后按 Enter 键进行安装即可，如图 1-11 所示（安装需要 10 分钟左右的时间），如果安装成功，在 Sublime Text→Preferences→Package Settings 中会看到 SublimeCodeIntel 项。

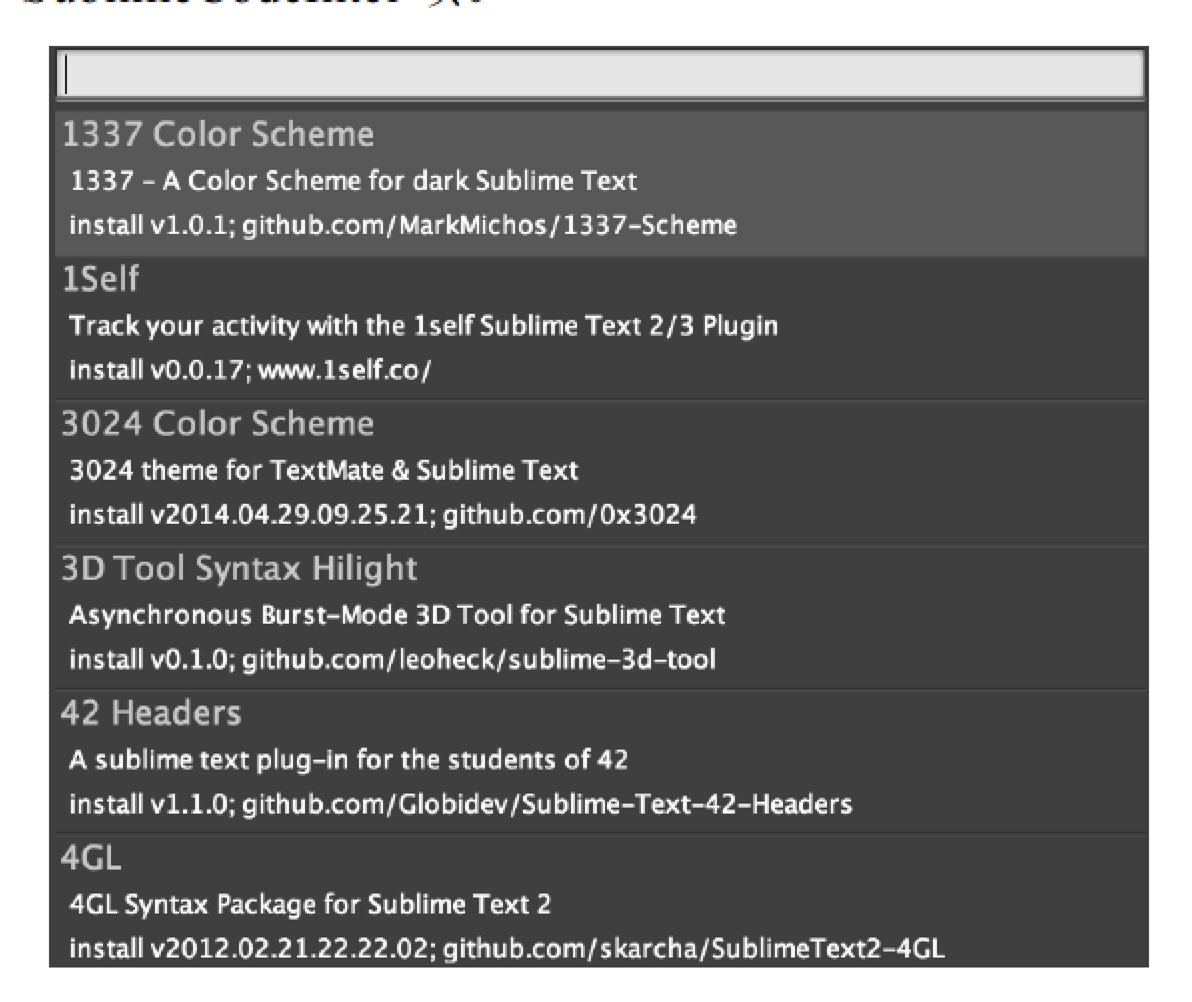


图 1-10 插件安装列表

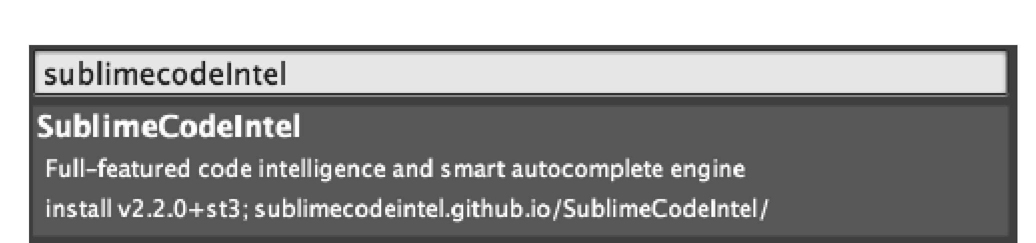


图 1-11 安装 SublimeCodeIntel 插件

在执行 install Package 命令时，Sublime Text 工具会从 PackageControl 官网拉取一个 JSON 文件，这个文件中是所有的 Sublime Text 插件信息，大小在数兆左右，不幸的是，这个文件的拉取由于国内网络情况依然会困难重重。笔者维护着一个此 JSON 文件的下载地址，无法成功拉取到文件的朋友可以尝试如下办法，通过 Sublime Text→Preferences→Package Settings→Package Control→Settings-User 打开用户配置文件，如图 1-12 所示。

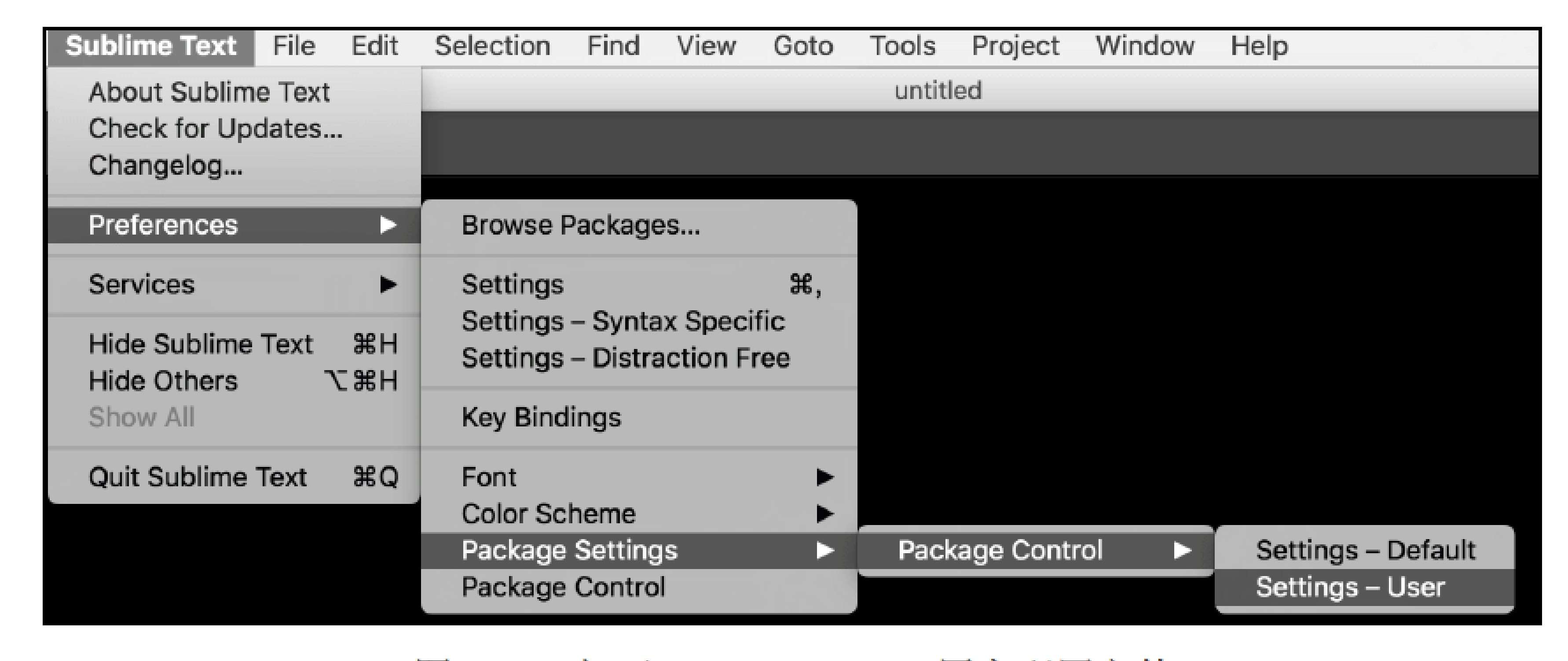


图 1-12 打开 Package Control 用户配置文件

在其中添加如下配置信息：

```
"channels": [
        "http://zyhshao.github.io/file/channel_v3.json"
    ],
```

添加完成后的配置文件看上去如图 1-13 所示。

```
{
    "bootstrapped": true,
    "installed_packages":
    [
        "Package Control"
    ],
    "channels": [
        "http://zyhshao.github.io/file/channel_v3.json"
    ],
}
```

图 1-13　进行用户配置文件设置

上面将插件列表 JSON 文件的拉取地址修改为了 http://zyhshao.github.io/file/channel_v3.json。

成功安装 SublimeCodeIntel 插件后，尝试编写一些 JavaScript 代码，可以看到 SublimeCodeIntel 插件的智能提示十分强大，如图 1-14 所示。

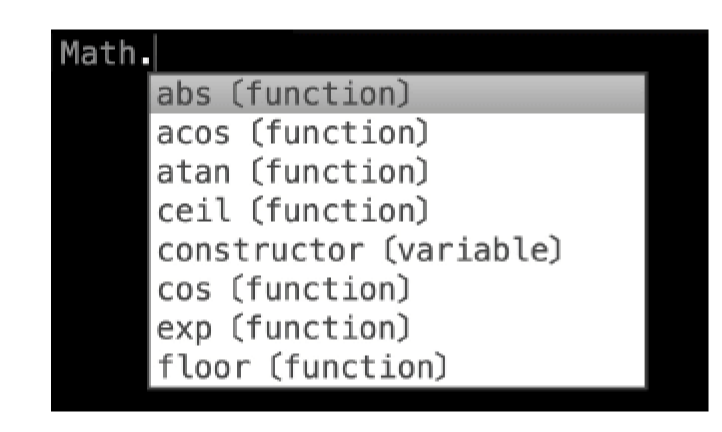


图 1-14　SublimeCodeIntel 的代码智能提示

1.1.5　安装 JavaScript 代码格式化插件

缩进规范的代码会使我们在编写程序时赏心悦目，在 PackageControl 的插件列表中输入“JsFormat”，按 Enter 键进行此插件的安装，如图 1-15 所示。

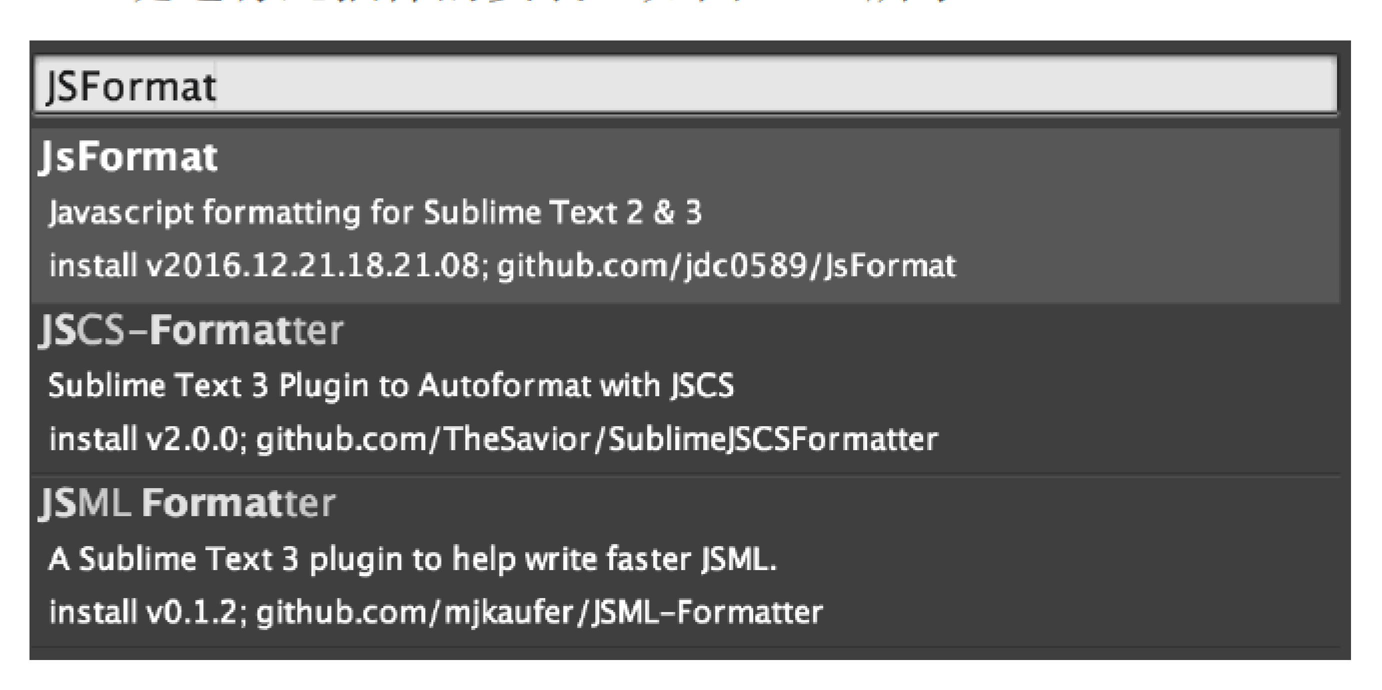


图 1-15　安装 JsFormat 插件

安装成功后，同样可以在 Sublime Text→Preferences→Package Settings 中看到 JsFormat 项。

JsFormat 插件的使用十分简单，选中要进行格式化的 JavaScript 代码，使用 Control+Option(alt)+F 即可对其进行格式化。

1.1.6　在 Sublime Text 中运行 JavaScript 代码

前边我们做了很多工作，安装了一些易用的 Sublime Text 插件，这些工具可以帮助我们更加愉悦地进行 JavaScript 代码的编写，代码的编写是为了运行，就以学习而言，能够实时看到代码运行的结果也会使学习效率大大提高。Sublime Text 自带支持对 Lua、Python、Ruby 等语言的编译运行功能，但是并不支持 JavaScript 语言，我们做一些额外的配置，来使 Sublime Text 支持编译运行 JavaScript 代码。

Node.js 是一款 JavaScript 运行时编译环境，并且也是运行 React Native 工程的基础环境，首先需要在系统中安装 Node.js 环境，从如下地址可以下载到安装包：https://nodejs.org/en/。

安装完成 Node.js 后，打开终端工具，在其中输入如下命令来查看 Node.js 的路径：

```
$ which node
```

打开 Sublime Text 工具，选择其中的 Tools→Build System→New Build System，如图 1-16 所示。在新建的文件中写入如下信息：

```
{
    "cmd": ["/usr/local/bin/node", "$file"],
    "selector": "source.js"
}
```

需要注意，上面的/user/local/bin/node 部分需要替换成你在终端中使用 which node 命令查找到的路径。之后将文件进行保存，命名为 JavaScript.sublime-build 即可。

新建一个 Sublime Text 文件，将其命名为后缀为 text.js。在 Tools→Build System 中将编译工具设为新创建的 JavaScript，编写一段 JavaScript 测试代码，使用 Command+B 进行代码的编译运行，可以看到，在 Sublime Text 控制台打印出代码的执行结果与所耗时间，如图 1-17 所示。

到此，已经配置完成了一款十分快速且强大的 JavaScript 代码学习工具，后面我们将使用 Sublime Text 来一步步打开 JavaScript 的编程世界，一起玩起来吧！

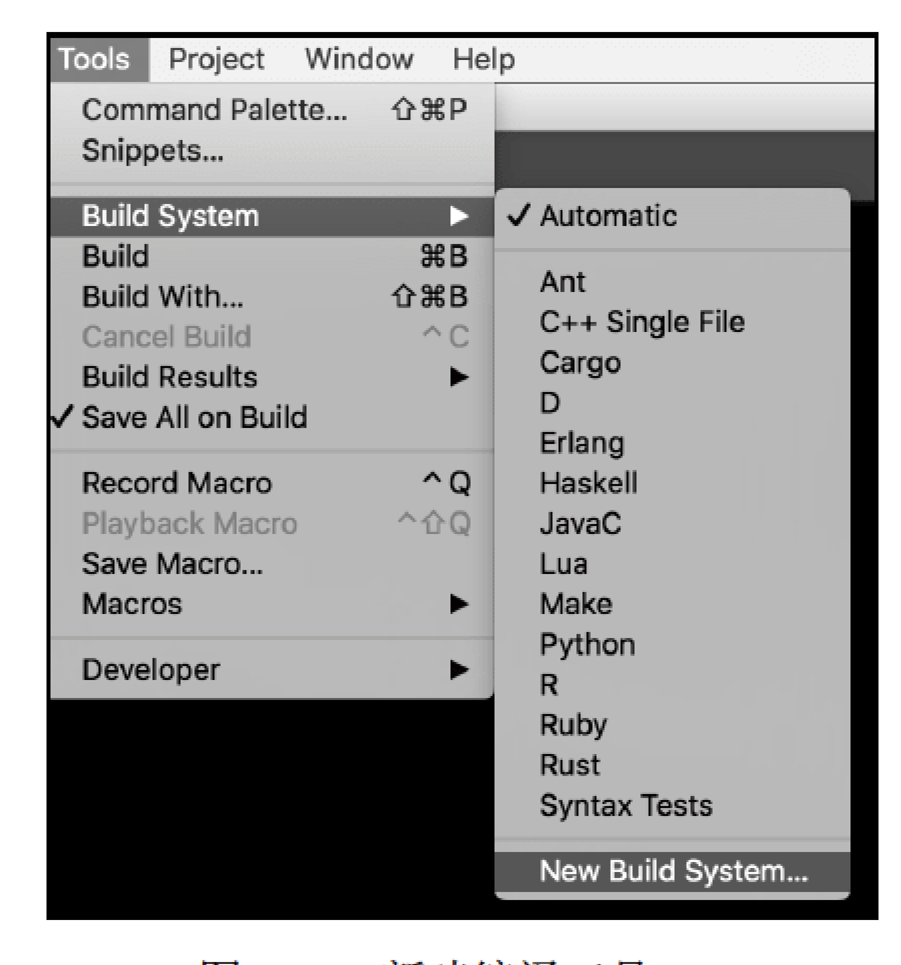


图 1-16 新建编译工具



图 1-17 在 Sublime Text 编辑器中进行 JavaScript 代码的执行

1.2 初识 JavaScript

JavaScript 虽然名字中包含 Java，但其和 Sun 公司的 Java 语言并没有太大的关系。JavaScript 是一种解释性的脚本语言，其设计的初衷是嵌入在 HTML 网页中进行使用，在 1995 年由 Netscape（网景）公司在浏览器上首先设计实现。JavaScript 实际上分为 3 个部分：ECMAScript 核心语法部

分，DOM 文档对象模型部分和 BOM 浏览器对象部分。本书我们主要使用 JavaScript 来进行 React Native 应用的开发，因此主要讨论 ECMAScript 核心语法部分。

博 闻 强 识

Netscape 公司最初将其设计的脚本语言命名为 LiveScript，后来为了使其迎合市面上非常流行的 Java 语言，也为了能使其看上去有些像 “Java”，将其修改为 JavaScript。微软公司为了取得技术优势，在同时期也推出了一种叫做 Jscript 的脚本语言来迎战 JavaScript。然而这样做的结果是使得开发者在编写前端网页时，要额外地考虑许多不同浏览器的兼容问题。为了达成规范，ECMA 国际（前身为欧洲计算机制造商协会）创建了 ECMA 标准，各家的脚本语言在语法上实际上都是对 ECMA 标准的实现。

1.2.1　JavaScript 的语法特点

JavaScript 是一种对大小写字母敏感的语言，也就是说不论变量名、函数名还是其他一切东西都是区分大小写的，例如下面的代码声明了两个完全不同的变量：

```
//大小写敏感
var name;
var NAME;
```

如果你熟悉一些编译型编程语言，例如 C++、Java、Swift，那么你可能会固执地认为所有的变量都要有强制的类型以确定其在内存中分配的空间大小，学习 JavaScript 时你需要忘记这条准则。JavaScript 中的变量是动态弱类型的，你可将一个变量先赋值为字符串类型的值再将其修改为数字类型的值，总之 JavaScript 中的变量没有特定的类型，你可以将其赋值为任意值。示例如下：

```
//动态类型
var dy = "string" //字符串
dy = 1  //数字
dy = true//布尔
```

抽 丝 剥 茧

虽然 JavaScript 允许你对一个变量进行多种类型值的赋值，但是在开发中尽量不要这样做，规范与固定意义的变量会使其他开发者赏心悦目。

博 闻 强 识

从原理上说，JavaScript 是编译型的，在编译后马上运行，并没有中间文件产生。

JavaScript 中每行结尾的分号可有可无，这一点十分类似 Swift 语言。注意，如果你在同一行中写了多条语句，就需要使用分号进行语句的分隔。这里强烈建议一行内只写一条语句，并且加上分号。示例代码如下：

```
//关于分号
var value1 = 1;
var value2 = 2; console.log(value2);
var value3 = 3 //行末尾可以省略分号
```

在 JavaScript 中，可以使用反斜杠进行字符串的折行编写，有时这样做可以使代码看起来更加漂亮，示例如下：

```
//使用反斜杠进行字符串的折行
var value4 = "\
Welcome to JavaScript \
My Good Friend!";
console.log(value4);
```

任何编程语言都会提供注释的能力，一个优秀的开发者不仅会写代码，更会写注释。JavaScript 中有两种方式进行注释的编写，使用双斜杠进行单行注释，使用双斜杠中间嵌入两个星号来进行多行注释，示例如下：

```
//这里是单行注释
/*
这里是
多行注释
*/
```

博 闻 强 识

有些开发者认为过多的注释并不是一件好事，优秀的程序代码应该有一定的字解释能力，这是正确的，但是关键的注释也是必要的，在决定写注释时，应该思考如下两点：

（1）如果不写注释，这里的代码功能是否一目了然，如果是，那就不要写注释。

（2）此处的代码是否会产生歧义并且不可优化，如果是，请一定要写注释。

1.2.2 JavaScript 中的变量

变量一词来源于数学，其往往代表函数中能够发生改变的量值。在计算机语言中，其是存储计算结构或表示值的抽象概念。需要注意，变量有可能是可变的，也有可能是不可变的，变量具体的意义由不同的编程语言所定义。在 JavaScript 中，使用 var 关键字来进行变量的声明（var 是 variable 单词的缩写）。

准确地说，JavaScript 中使用 var 关键字与变量名来声明或定义变量。示例如下：

```
//变量的声明与定义
var a;//声明变量 a
var b = 1; //定义变量 b
```

博 闻 强 识

关于声明与定义的区别，这里有些咬文嚼字，借用经典中的经典，谭浩强老师编写的标准本科教学书籍《C 程序设计》中的一句话：一般情况，把建立存储空间的声明称为定义，而把不需要建立存储空间的声明称为声明！

JavaScript 也允许在一条语句中同时声明或定义多个变量，使用逗号进行分割即可，其中变量的类型可以都不相同。示例如下：

```
//多个变量同时声明
var v1,v2=2,v3="hello";
```

在对变量进行命名时，有两条法则必须遵守：

（1）变量名的第 1 个字符必须是字母、下画线或者美元符号。
（2）除了第 1 个字符之外，余下的字符可以是下画线、美元符号或者任意数字与字母。

下面这些变量名都是合法的：

```
var _myName_,MyName,$name,_3name,n3;
```

下面这些变量名都是非法的：

```
//不合法的变量名
var 31;
var %2;
```

虽然 JavaScript 对变量的命名比较自由，但并不意味着你在命名变量时可以随心所欲。正确的变量命名应该能够做到见形知意，并且从外观上看起来突兀自然。比较著名的变量命名方法有如下几种：

（1）Camel（驼峰）命名法。Camel 命名法是指变量的首字母小写，接下来的每个单词的首字母大写，示例如下：

```
//驼峰命名法
var myName;
```

（2）Pascal 命名法。Pascal 命名法是指变量的首字母大写，其后每个单词的首字母也大写。Pascal 命名法有时也被称为大驼峰命名法，示例如下：

```
//Pascal 命名法
var MyName;
```

（3）匈牙利类型命名法。Camel 与 Pascal 命名法只针对变量的意义进行解释，匈牙利类型命名法中还加入了变量的类型，其规则是在 Pascal 命名法的基础上在变量名的最前边加上变量类型的标识，例如数字型变量添加 i 标识，字符串变量添加 s 标识，示例如下：

```
//匈牙利类型命名法
var iAge = 25;
var sName = 'jaki';
```

表 1-1 列出了常用类型对应的标识。

表 1-1 变量类型及对应的标识

类型	标识	类型	标识
数组	a	对象	o
布尔	b	正则表达式	re
浮点数字	f	字符串	s
整数	i	任意类型	v
函数	fn	对象	o

另外，对于一些大小写不敏感的编程语言，也常常采用下画线命名法，即单词与单词之间使用下画线进行分割，示例如下：

```
//下画线命名法
var my_name;
```

JavaScript 中的变量还有一个十分有意思的特点，注意，如果你有其他编程语言的基础，这点可能会颠覆你的经验。JavaScript 中变量不一定都要使用 var 关键字进行声明，如下的代码依然可以很好地执行：

```
//未声明的变量
unKnow = "this is a not declare variable"
console.log(unKnow);
```

JavaScript 解释器遇到上面的代码后会自动声明一个全局的变量 unKnow，在此郑重提醒，这个特点是 JavaScript 的便利之处，但也十分危险，在复杂程序中，这样做也会使变量的追踪格外困难，代码虽然是让计算机执行的，但却是让人看的，编写代码的过程本身就是一种艺术，最优雅的方式是将变量声明在其使用之前。

1.3 JavaScript 中的数据类型

变量用来存储特定意义的值。在 JavaScript 中，变量可以存储两种类型的值：原始值和引用值。原始值和引用值的区别在于存储的位置与访问的方式不同。原始值是存储在栈中的简单数据，引用值是存储在堆中的对象数据。这也就是说，当你通过变量名访问原始值时，会直接访问到其存在栈中的数据，而在使用变量名访问引用值时，会首先获取到其存在栈中的对象地址，根据地址再向堆中查找真正的对象数据。JavaScript 中定义的原始类型有 5 种，分别为 Undefined（未定义类型）、Null（空对象类型）、Boolean（布尔类型）、Number（数字类型）、String（字符串）类型。图 1-18 中描述了原始值与引用值的差异。

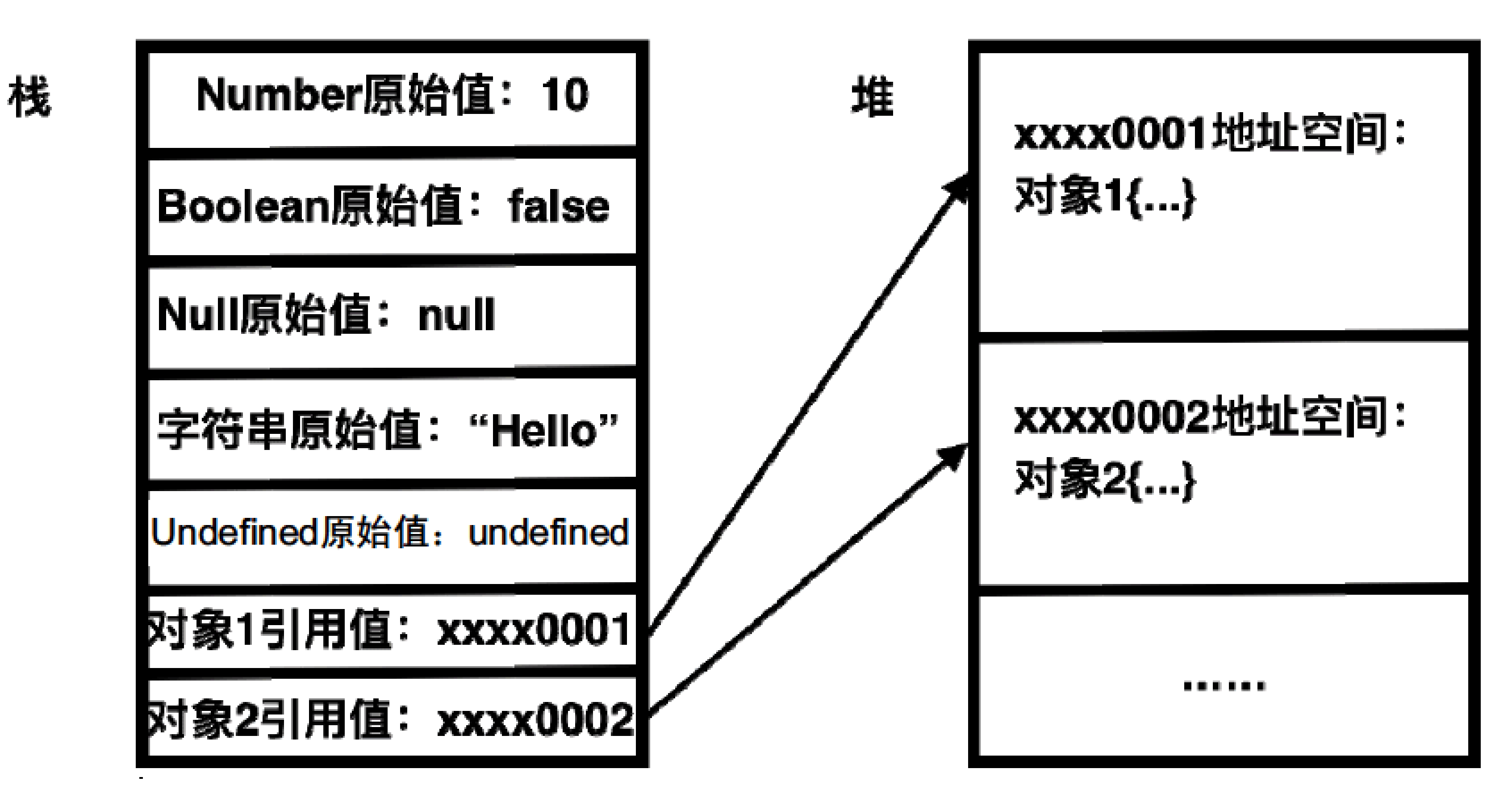


图 1-18　原始值与引用值图示

抽 丝 剥 茧

原始值所占的内存大小一般是固定不变的，将其存储进栈中可以更快地进行数据访问，引用值所占的内存大小通常较大并且不固定，但其地址的大小是固定不变的，将其地址存入栈中不会带来任何性能影响。在 5 种原始类型中，String 类型十分特殊，因为其大小也是不固定的，在 Java、Objective-C 等语言中，字符串通常会被定义为引用类型，但 JavaScript 中依然将其作为一种原始类型。

1.3.1　原始类型

JavaScript 中定义的原始类型有 5 种：Undefined、Null、Boolean、Number、String。切记，只有这 5 种原始类型。原始类型数据的创建是直接使用字面值来创建的。

Undefined 类型只有一个值，即 undefined。其意义也如其所描述的那样，它代表的是未定义。例如，一个变量只是被声明，其值就是 undefined，类型就是 Undefined 类型，JavaScript 中的 typeof 关键字可以用来获取变量或值的类型，示例如下：

```
//Undefined 类型
var unKnown;
console.log(typeof unKnown);//将打印 undefined
```

注意，仅仅被声明而未赋值的变量是未定义的，实际上从没有声明过的变量也是未定义的，示例如下：

```
//未声明过的变量也是未定义的
console.log(typeof unKnown2);//将打印 undefined
```

抽 丝 剥 茧

typeof 是一个特殊的运算符，如果将未声明过的变量用于其他运算符中将会产生运行错误。

执行一个没有返回值的函数后，也会返回 undefined 值，示例如下：

```
//一个无返回值的函数
function func(){
    console.log("func");
}
var v1 = func();//将打印 func
console.log(v1);//将打印 undefined
```

可能你现在对函数还不太理解，不用担心，后面我们会专门学习函数的相关内容。

Null 类型是 JavaScript 中另一种只有一个值的类型，其字面值为 null。Null 类型的定义唯一的用途是作为空对象的占位。现在，你可能对对象也不太理解，在我们介绍原始值与引用值的时候已提到对象，对象是一种复杂数据类型，对象变量中实际存储的是对象的引用地址，JavaScript 中对象不属于原始类型，原则上说他们之间并不会产生强关系，然而在实际开发中，开发者往往需要一种约定的值来表示对象为空，即要有一个约定的值来描述一个无用的地址，这个值就是 null，当开发者在使用对象前，发现变量中存储的引用地址为 null 时，就可以知道此对象还没有被初始化，或者此对象已经不存在了。

如果在 JavaScript 中使用 typeof 来对 null 值进行类型检查，你会惊奇地发现其返回的类型是 object，这或许是 JavaScript 前期实现上的一个错误，但是其也恰恰与 null 是空对象的占位这一概念完全切合。因此，在 ECMAScript 标准中沿用了这一定义。示例如下：

```
//Null 类型
var obj = null;
console.log(typeof obj);        //将打印 object
```

博 闻 强 识

现实中的很多设计可能都不是最正确的，却无疑是最合适的。一个很有趣的例子来自现代的键盘设计，明确来说，现在的键盘布局并不科学，这种“QWERTY”布局的键盘最开始的设计是为了打字机所使用，正常排序的键盘在打字机打字速度过快时，往往会产生卡顿问题。为了解决这个问题，克里斯托夫 · 拉森 · 肖尔斯刻意将高频字符放置在相反的方向，以最大限度地放慢敲键速度。这也就是说，现代键盘的布局设计完全是为了更慢速度地打字。另一种更科学的键盘布局方式为 Dvorak 布局，如果在互联网上搜索 Dvorak 关键字，你会搜出一万种理由阐述这种布局的好处，然而其依旧无法成为主流，人们的习惯根深蒂固并且不会尝试去改变！

Boolean 类型中定义了两个值，分别为 true 与 false。其在逻辑运算中应用广泛。

Number 类型用来描述数字，和其他编程语言不同的是，JavaScript 中的 Number 类型既可以描述整数值，也可以描述浮点值。示例如下：

```
//Number 类型
var num1 = 100//整数值
var num2 = 3.14//浮点值
console.log(typeof num1);//将打印 number
console.log(typeof num2);//将打印 number
```

在数值前添加前缀，可以将其描述为八进制或十六进制的数值，八进制需要添加 0 作为前缀，十六进制需要添加 0x 作为前缀，示例如下：

```
//八进制
var num3 = 011;//对应十进制 9
//十六进制
var num4 = 0x11;//对应十进制 17
```

抽 丝 剥 茧

需要注意，很多编程语言并不介意数值量前面是否添加多余的 0，JavaScript 语言对这一点要求十分严格，多余的 0 会改变数值的进制方式，造成不可控的错误。

对于非常大或非常小的数值，JavaScript 中也可以使用科学计数法来对其进行描述，使用字母 e 来描述 10 的 e 次方，示例如下：

```
//科学计数法
var num5 = 1.01e3;//对应 1010
var num6 = 1111000e-6;//对应 1.111
```

JavaScript 中还定义了一些特殊的数值，比如 Number.MAX_VALUE 和 Number.MIN_VALUE 分别表示 Number 类型所能表示的最大值与最小值，示例如下：

```
//Number 最大可以表示的值
console.log(Number.MAX_VALUE);//打印 1.7976931348623157e+308
//Number 最小可以表示的值
console.log(Number.MIN_VALUE);//打印 5e-324
```

当计算值超出了 Number 类型所能表示的极限时，即会被认为是所谓的无穷。JavaScript 中也专门定义了特殊的 Number 值来表示无穷，其中 Number.POSITIVE_INFINITY 表示正无穷大，Number.NEGATIVE_INFINITY 表示负无穷大，它们的值分别为 Infinity 与-Infinity，示例如下：

```
//正无穷
console.log(Number.POSITIVE_INFINITY);//将打印 Infinity
//负无穷
console.log(Number.NEGATIVE_INFINITY);//将打印-Infinity
```

JavaScript 中定义的最后一个比较特殊的 Number 值为 NaN，是 not a Number 的缩写，表示不是一个数字。这个值在字符串向数字转换失败时被返回，示例如下：

```
//NaN 值
var num7 = Number("w");
console.log(num7);//将打印 NaN
```

需要注意，NaN 十分特殊，既不可以进行计算也不可以进行比较，并且与自身也不相等，例如如下的比较将会返回 false：

```
console.log(NaN == NaN);//将打印 false
```

如果要判断一个变量的值是否是 NaN，可以使用如下方法：

```
console.log(isNaN(num7));//将返回 true
```

String 类型是 JavaScript 中唯一没有固定大小的原始类型，用来存储多个 Unicode 字符。在 C、Java 这些语言中，字符和字符串是两种不同的类型，字符使用单引号包括，字符串则使用双引号包括。在 JavaScript 中，删去了字符的概念，字符串既可以使用单引号包括也可以使用双引号包括，但是如果要在字符串中嵌套字符串，则单双引号必须交替使用。示例如下：

```
//String 类型
var str1 = "Hello";
var str2 = 'World';
var str3 = "Hello 'World'";
```

和 C、Swift、Java、Perl 这些语言类似，JavaScript 中也定义了一些转义字符，可参见表 1-2。

表 1-2　转义字符及含义

转义字符	含义	转义字符	含义
\n	换行	\'	单引号
\t	制表符	\"	双引号
\b	空格	\0nnn	使用八进制代码表示字符
\r	回车	\xnn	使用十六进制代码表示字符
\f	换页符	\unnnn	使用十六进制 Unicode 码表示字符
\\	反斜杠		

1.3.2　引用类型

JavaScript 中的引用类型数据实际上指的是对象。对象是一组功能行为互补的数据集合，其用来模拟实现生活中的事物。举一个简单的例子，如果要开发一款教学系统软件，这个系统中需要包含老师和学生两类成员，在这个软件中，老师就是一种对象，学生也是一种对象。老师对象中可能会包含姓名、教师编号、专业、所带班级等，学生对象中可能会包含姓名、年龄、所学课程、所在班级等。当然，除了这些描述对象属性信息的数据外，对象中还需要包含一些行为，例如老师要能够进行教学任务、学生要学习考试等。在 JavaScript 语言中，Object 类型就是这样的一种引用类型，其创建出来的实例被称为对象。

有两种方式来进行对象的创建，可以直接通过 Object 构造方法来新建对象，以教师对象为例，代码如下：

```
//创建教师对象
var teacher = new Object();
//为教师对象添加一些属性
teacher.name = '珲少';
teacher.age = 25;
teacher.subject = 'JavaScript';
//为教师对象添加行为方法
```

```
teacher.teach = function(){
    console.log('正在进行教学...');
};
```

上面的代码为教师对象添加了姓名、年龄和所教科目的属性，并且为其添加了一个教学行为 teach 方法。我们也可以直接通过字面值的方式来创建教师对象，示例如下：

```
//字面值直接创建对象
var teacher2 = {
    name:'Jaki',
    age:25,
    teach:function(){
        console.log('正在进行教学...');
    }
};
```

如果将 teacher 对象与 teacher2 对象的类型都进行打印，可以看到它们都是 Object 类型：

```
console.log(teacher);//{ name: '珲少', age: 25, subject: 'JavaScript', teach:
[Function] }
console.log(teacher2);//{ name: 'Jaki', age: 25, teach: [Function] }
console.log(typeof teacher);//object
console.log(typeof teacher2);//object
```

要使用对象的某个属性，有两种方式来获取。比较方便且常用的方式是点语法取值，示例如下：

```
//取对象的属性
console.log(teacher.name);//珲少
console.log(teacher.age);//25
console.log(teacher.subject);//JavaScript
```

另外，也可以通过键名字符串的方式来获取对象的属性，示例如下：

```
//通过键名字符串取值
console.log(teacher['name']);//珲少
```

需要注意，通过键名字符串的方式来取对象的属性时，所传入的键必须是字符串类型的。

对象中定义的函数是用来描述对象的行为的，同样可以使用点语法来使对象执行行为，示例如下：

```
//让对象执行行为
teacher.teach();//将打印 正在进行教学...
```

同样，使用键名获取到的方法也可以执行行为：

```
teacher['teach']();//将打印 正在进行教学...
```

现在你可以简单地理解了，在面向对象的语言世界里，万事万物基本上都是对象（并不是全部），简单的对象组合成复杂的对象，复杂的对象协作完成复杂的功能。在本节，我们先不对“对象”做深入的研究，后面的章节会介绍更加复杂的面向对象机制。

1.4　JavaScript 中的运算符

运算符用来执行程序代码的运算。一个完整的表达式应该由两部分组成，分别为操作数与运算符，例如简单的加法表达式“1+2”中，数字“1”和数字“2”都是操作数，符号“+”是加法运算符，其作用是将前后两个操作数进行加法运算。

在 JavaScript 中，运算符可以分为如下几类：

（1）算术运算符。
（2）赋值运算符。
（3）关系运算符。
（4）逻辑运算符。
（5）位运算符。
（6）自增、自减、条件、逗号等特殊运算符。

本节将向你介绍这些运算符的用法。

1.4.1　算术运算符

算术运算符用来做常用的数学运算，例如加、减、乘、除等。符号“+”是 JavaScript 中的加法运算符，数字或者字符串都可以使用其来进行相加操作，对应的减法运算符为符号“-”，示例如下：

```
//加法运算符
var sum = 5+5;
console.log(sum);//10
var newString = "Hello" + " World";
console.log(newString);//Hello World
```

你可能会有疑问，在 JavaScript 中定义了许多特殊的数值，例如 NaN 与 Infinity，如果在加法运算的操作数中有这些值，会运算出什么样的结果呢？其实在 JavaScript 中，这些特殊值的运算方式和生活中的场景基本一致。有如下几个规则：

（1）如果操作数中有一个是 NaN，则运算结果为 NaN。
（2）正无穷值与正无穷值进行加法运算，结果为正无穷值。
（3）负无穷与负无穷值进行加法运算，结果为负无穷值。
（4）如果进行数值与字符串的相加操作，结果会被强制转换成字符串。

示例代码如下：

```
//加法运算中的几个特殊规则
console.log(1+NaN);//NaN
```

```
console.log(Infinity+Infinity);//Infinity
console.log(-Infinity + -Infinity)//-Infinity
console.log(1+'1');//11
```

在数学中与加法互为逆运算的为减法，JavaScript 中使用符号“-”来进行减法运算，示例如下：

```
//减法运算符
var sub = 10-5;
console.log(sub);//5
```

减法运算符有一点非常特殊，如果其进行计算的两个操作数中有字符串类型，这个字符串类型的数据可以转换为数字，则 JavaScript 会自动帮其转换成数字后再进行减法运算，如果此字符串数据不能转换为数字，则会计算出 NaN 结果，示例如下：

```
console.log("10"-5);//5
console.log("10"-"3");//7
console.log("s"-3);//NaN
console.log("10"-"a");//NaN
```

针对一些特殊值的减法运算，JavaScript 中也定义了一些规则：

（1）如果某个操作数是 NaN，则运算的结果为 NaN。
（2）正无穷值减去正无穷值，结果为 NaN。
（3）负无穷值减去负无穷值，结果为 NaN。
（4）正无穷值减去负无穷值，结果为正无穷值。
（5）负无穷值减去正无穷值，结果为负无穷值。

示例如下：

```
//减法运算中的几个特殊规则
console.log(1-NaN);//NaN
console.log(Infinity-Infinity);//NaN
console.log(-Infinity - -Infinity);//NaN
console.log(Infinity - -Infinity);//Infinity
console.log(-Infinity - Infinity);//-Infinity
```

当符号“+”与符号“-”作为一元运算符时，其便成了正号运算符与负号运算符。对数字进行正号或负号运算时，正号运算会保持数字的正负性，负号运算会改变数字的正负性，示例如下：

```
console.log(+num1);//不改变符号 10
console.log(+num2);//不改变符号 -10
console.log(-num1);//改变符号 -10
console.log(-num2);//改变符号 10
```

抽 丝 剥 茧

正负运算符还有一个很实际的用途是可以将字符串值强制转成数字值，这在开发中十分常用。示例如下：

```
console.log(typeof +"1")//number
```

JavaScript 中使用乘法运算符“*”来进行乘法运算，示例如下：

```
//乘法运算符
var mul = 3*4;
console.log(mul);//12
```

对于乘法运算，也存在下面几条特殊的规则：

（1）如果某个操作数是 NaN，则结果为 NaN。
（2）无穷值乘以 0，结果为 NaN。
（3）无穷值乘以 0 以外的其他数字，结果为无穷值。
（4）无穷值乘以无穷值，结果为无穷值。

示例代码如下：

```
//乘法运算中的几个特殊规则
console.log(1*NaN);//NaN
console.log(Infinity*0);//NaN
console.log(Infinity * 1);//Infinity
console.log(Infinity * -1);//-Infinity
console.log(Infinity * Infinity);//Infinity
console.log(-Infinity * -Infinity);//Infinity
console.log(-Infinity * Infinity);//-Infinity
```

运算符“/”在 JavaScript 中用来进行除法运算，示例如下：

```
//除法运算符
var del = 88/10;
console.log(del);//8.8
```

除法运算符对于特殊值运算的规则如下：

（1）如果某个操作数是 NaN，则结果为 NaN。
（2）无穷值除以无穷值，结果为 NaN。
（3）无穷值除以任何数，结果为无穷值。
（4）任何数除以无穷值，结果为 0。
（5）任何数除以 0，结果为无穷值。
（6）0 除以任何数，结果为 0。

示例代码如下：

```
//除法运算中的几个特殊规则
console.log(10/NaN);//NaN
console.log(Infinity/Infinity);//NaN
console.log(Infinity/100);//Infinity
console.log(10/Infinity);//0
console.log(100/0);//Infinity
console.log(0/100);//0
```

JavaScript 中还支持进行求余数的运算，取余运算也叫取模运算。符号“%”为取模运算符。示例如下：

```
//取模运算符
var res = 17%8;
console.log(res);//1
var res2 = 10.7%1.5
console.log(res2);//约等 0.2
```

对于特殊值的取模运算，也有如下规则：

（1）无穷值对任何值取模结果都是 NaN。
（2）非无穷值对无穷值取模结果为非无穷值本身。
（3）0 对任何数取模结果都是 0。
（4）任何数对 0 取模结果都是 NaN。

示例代码如下：

```
//取模运算中的几个特殊规则
console.log(Infinity%1);//NaN
console.log(Infinity%Infinity);//NaN
console.log(100%Infinity);//100
console.log(0%100);//0
console.log(100%0);//NaN
```

博 闻 强 识

在很多编程语言中，取模运算都不可以以浮点数作为操作数。在 Swift 语言的早期版本中是支持浮点数的取模运算的，在 Swift3 之后的版本中又将这个特性删去了，JavaScript 是一种相对特殊的语言，并没有对浮点数取模运算做太严格的控制。

1.4.2　赋值运算符

赋值运算符的作用是将表达式的值赋给变量。从接触到 JavaScript 语言开始，我们就一直在使用赋值运算符，最简单的赋值运算符使用示例如下：

```
//赋值运算符
var string = "Hello World";
```

JavaScript 中还提供了一些复合赋值运算符，示例如下：

```
//复合运算符
//复合加赋值运算符
var v1 = 0;
v1+=10;   //相当于 v1=v1+10;
console.log(v1); //10
v1-=9;    //相当于 v1=v1-9;
```

```
console.log(v1);//1
v1*=10;   //相当于 v1=v1*10;
console.log(v1);//10
v1/=10;   //相当于 v1=v1/10;
console.log(v1);//1
v1&=0;    //相当于 v1=v1&0;
console.log(v1);//0
v1|=1;    //相当于 v1=v1|1;
console.log(v1);//1
v1<<=1;   //相当于 v1=v1<<1;
console.log(v1);//2
v1>>=1;   //相当于 v1=v1>>1;
console.log(v1);//1
v1>>>=1;   //相当于 v1=v1>>>1;
console.log(v1);
```

其中，“&”“|”“<<”等符号看上去可能有些陌生，这些是 JavaScript 中的位运算符，后面会专门介绍位运算的相关知识，这里你只需要记住，复合赋值运算符实际上是将一个变量作为操作数经过计算后再次赋值给它自身。

1.4.3 关系运算符

在代码的编写过程中，比较操作十分常用，例如比较两个字符串是否相同、比较两个数字的大小等。比较运算符的计算结果将会返回一个布尔值，通过布尔值的真或假可以实现不同的业务逻辑。

数字之间的比较是最常规的比较，示例如下：

```
//数字之间的比较
console.log(1<2);//true
console.log(1>2);//false
console.log(1==2);//false
```

符号“<”为小于运算符，当第 1 个操作数小于第 2 个操作数时结果为 true，否则结果为 false。“>”为大于运算符，当第 1 个操作数大于第 2 个操作数时结果为 true，否则结果为 false。需要额外注意，在 JavaScript 中，等于运算符用符号“==”来表示，这和数学常识有些差异，编程初学者十分容易混淆。

比较运算符也可以在字符串与字符串间进行比较操作，字符串的比较遵守这样的法则：逐字符进行字符码大小的比较，如果字符码相同，则比较下一字符直到比较出结果或者比较完成所有字符。示例如下：

```
//字符串与字符串进行比较
console.log("a">"b");//false
console.log("a"<"b");//true
console.log("ss"=="ss");//true
```

需要注意，如果是描述数字的字符串进行比较，依然会遵守上面所说的法则，例如下面的比较：

```
console.log("12">"3");//false
```

"12">"3"的比较结果返回的是 false，这是正确的，因为 JavaScript 解释程序会将字符“1”与字符“3”的字符码进行比较，将结果返回。

数字和字符串进行比较相对要棘手一些。如果是描述数字的字符串与数字进行比较，JavaScript 解释程序会将字符串强制转换成数字类型后再进行比较，例如：

```
console.log("3">10);//false
```

如果是非数字的字符串与数字进行比较，结果将永远是 false，例如：

```
console.log("a">0);//false
```

JavaScript 中还可以使用“>=”与“<=”来进行不小于和不大于的比较运算，其规则和“>”符号与“<”符号一致。示例如下：

```
console.log(12<=12);//true
console.log(1>=2);//false
```

关于等于与不等于的比较在 JavaScript 中有两类，一种是相等比较“==”与不相等比较“!=”，另一种是全等比较“===”与不全等比较“!==”。全等比较运算并非是 JavaScript 语言所独有的，许多编程语言中都有类似的运算符，例如 Swift。

在进行相等或不相等比较时，不同类型间数据的比较遵守如下几条原则：

（1）布尔值在比较运算前会被转换成数值，true 转换成 1，false 转换成 0。
（2）描述数字的字符串在与数字进行比较前会被转换成数字。
（3）对象和字符串进行比较前，会将对象转换成字符串。
（4）null 值和 undefined 值进行相等比较时结果为 true。

示例代码如下：

```
console.log(true==1);//true
console.log(2==true);//false
console.log(false==0);//true
console.log("11"==11);//true
var obj = {name:'jaki'};
console.log(obj=="[object Object]");//true
console.log(1!=2);//true
console.log(null==undefined);//true
```

需要注意，如果进行比较操作的是引用值而非原始值，则比较的实际是所引用对象的地址。

抽 丝 剥 茧
再次提醒，NaN 与 NaN 进行相等比较，结果是 false。

“==”与“!=”运算在进行比较前会根据上面的规则对操作数进行类型的转换，全等运算符“===”与不全等运算符“!==”在比较前不会做任何类型转换。换句话说，全等和不全等进行比较时，既会比较类型，也会比较值，只有类型和值完全相等的两个操作数才被认为是全等。示例如下：

```
//全等比较
console.log(11==="11");//false
console.log(true!==1);//true
```

博 闻 强 识

有一个很有趣的小例子，在 JavaScript 中，如果下面的代码输出 false:

```
console.log(ex>1);
```

那么如下代码将一定输出 true 么？

```
console.log(ex<=1);
```

答案是否定的，如果 ex 变量在进行比较转换时被转换成了 NaN，则上面两句输出的都将是 false:

```
var ex = "ss";
console.log(ex>1);//false
console.log(ex<=1);//false
```

1.4.4 逻辑运算符

逻辑运算对于一门编程语言至关重要，它是分支和循环结构的基础，JavaScript 中支持的逻辑运算有 3 种：逻辑与运算、逻辑或运算和逻辑非运算。

JavaScript 中使用符号“&&”进行逻辑与运算。逻辑运算通常进行在两个布尔类型的操作数之间，与运算需要遵守如表 1-3 所示的运算规则。

表 1-3 与运算

操作数 1	操作数 2	结果
true	true	true
true	false	false
false	true	false
false	false	false

上面的运算规则可以简要概述为：进行逻辑与运算的两个操作数都为 true，结果才为 true，其中有一个操作数为 false，结果就为 false。

在有些强类型的编程语言中，逻辑运算符只能在布尔值之间进行运算，JavaScript 中逻辑与运算的操作数可以是任意类型的，并且其运算结果也不一定是布尔类型的值，其规定如下：

（1）在两个对象间进行逻辑与运算，结果将返回第二个对象。
（2）在进行逻辑与运算的两个操作数中，如果有一个操作数为 null，则结果为 null。
（3）在进行逻辑与运算的两个操作数中，如果有一个操作数为 NaN，则结果为 NaN。
（4）在进行逻辑与运算的两个操作数中，如果有一个操作数为 undefined，则结果为 undefined。

示例代码如下：

```
//与运算的相关规则
var obj = {name:'jaki'};
//两个对象进行逻辑与运算 结果为第二个对象
console.log({}&&obj);
console.log(null&&true);//null
console.log(true&&null);//null
console.log(NaN&&true);//NaN
console.log(true&&NaN);//NaN
console.log(undefined&&true);//undefined
console.log(true&&undefined);//undefined
```

博 闻 强 识

先来看一个十分有趣的小例子：

```
var v1 = 10;
var v2 = true;
console.log(v2&&(v1++));
console.log(v1);
var v3 = 10;
var v4 = false;
console.log(v4&&(v3++));
console.log(v3);
```

你能猜出上面代码中的 console.log(v1)与 console.log(v3)分别会打印出什么样的结果么？结果是 console.log(v1)将打印 11 而 console.log(v3)将打印 10。

对上面的结果是不是有些许意外？其实很多编程语言在处理逻辑运算时都有这样一种法则：如果第一个操作数已经可以确定此表达式的结果，则不会再执行第二个操作数，以上面的代码为例，v4 为 false 时已经可以确定此与运算结果为 false，因此 v3++将不会被执行到。

JavaScript 中使用符号“||”进行逻辑或运算。逻辑或运算遵守如表 1-4 所示的运算法则。

表 1-4　或运算

操作数 1	操作数 2	结果
true	true	true
true	false	true
false	true	true
false	false	false

和逻辑与运算类似，JavaScript 中的逻辑或运算也不一定会返回逻辑值。规定如下：

（1）如果有一个操作数为对象，当对象为第 1 个操作数时，结果为对象本身；当对象为第 2 个操作数时，如果第 1 个操作数为 false，则结果为对象本身，如果第 1 个操作数为 true，则结果为 true。

（2）如果两个操作数都为对象，则会返回第一个操作数。

示例代码如下：

```
//或运算规则
console.log(obj||false);//obj
console.log(true||obj);//true
console.log({}||obj);//{}
```

同样，如果在进行逻辑或运算时，第一个操作数已经可以决定表达式的值，则不会再执行到第二个操作数处。

JavaScript 中的逻辑非运算使用“！”符号定义。需要注意，逻辑非运算一定会返回布尔值。逻辑非运算也被称为逻辑取反运算，其遵守如表 1-5 所示的规则。

表 1-5 非运逄

操作数	结果
true	false
false	true

当操作数不全是逻辑值时，有如下规则：

（1）如果操作数是对象，则返回 false。
（2）如果操作数是数字 0，则返回 true。
（3）如果操作数是非 0 数字，则返回 false。
（4）如果操作数是 null，则返回 true。
（5）如果操作数是 NaN，则返回 true。
（6）如果操作数是 undefined，则返回 true。

1.4.5 位运算符

我们知道，程序中的所有数在计算机内存中都是以二进制的形式存储的。这也很好理解，进制的实质是确定计数时逢几进一，人类有 10 根手指，因此很久以前的祖先开始就习惯了使用十进制来计数。计算机的核心是由电子元件组成的，而电子元件最容易描述的两种状态便是高电平与低电平，因此使用二进制计数是最安全、最便捷的方式。

在介绍 JavaScript 中的位运算前，先来简单地了解一下 JavaScript 中的二进制计数。JavaScript 中只有一种数值类型：原始类型 Number。但是实际上，JavaScript 中存储的数值有两种，分别为有符号数和无符号数。其实这和大多数编程语言类似，只是有些强类型的语言会将数值类型再进行细化，比如 8 位整型、32 位整型、64 位整型、32 位浮点型或 64 位浮点型。JavaScript 中所有的数值默认都是 32 位的（当然这样说并不准确，具体的位数和计算机环境有关，目前大多都是 32 位的）。位数可以简单理解为表示一个数字需要多少个二进制位，我们暂定 JavaScript 中所有的数值都是 32 位的，那么 8 这个十进制数在内存中存储的数据如下：

0	0	……	…….	0	0	1	0	0	0

上面每一个方格表示一个二进制位，中间的省略号代表省略中间的 0，一共 32 个小方格，代表 32 个数位。

JavaScript 中所有的数值创建时默认都是有符号的，虽然存储一个数值需要 32 位，我们能够操作的实际上只有 31 位，最后一位作为符号位，符号位为 0 表示这个数值是正数，符号位为 1 表示这个数值是负数。对于正数，存储在内存中的二进制数据很好理解，求得此正数的二进制形式放入内存，无用的位补零即可。对于负数，其在内存中存储的是二进制补码方式，计算补码的步骤如下：

步骤 01 确定该数的绝对值的二进制形式。

步骤 02 对此二进制码求反码（0 和 1 互相交替）。

步骤 03 在反码的基础上加 1。

根据上面的规则，以十进制数-8 为例，其绝对值的二进制形式为 0…01000，对其求反码为 1…10111，在其基础上再加 1 得到 1…11000，即十进制数-8 实际存在内存中的数据如下：

1	1	……	……	1	1	1	0	0	0

相对于有符号数，无符号数中并没有负数，所有的数值都是正数，在这种情况下，正负位就失去了作用，因此对于无符号数来说，其 32 个二进制位都是用来表示数字的。

博 闻 强 识

计算机中为什么要采用补码的方式来存储数据？对于有符号数，最高位表示的是符号，如果直接进行二进制形式的存储，难免会出现这样一种情况：0 可以表示为正数 0 和负数 0，这有悖现实规律，因此人们采用补码的方式来使现实的数值与计算机内存中存储的二进制数据一一对应，正数的补码是其本身，负数的补码是其反码加 1，经过这样的计算后，无论正数 0 还是负数 0 在计算机内存中存储的都是全 0 码，做到了统一。

JavaScript 中的 Number 数据可以调用 toString()方法来将其转换成字符串，这个方法中可以传入一个参数设置要转换为数值的进制。示例如下：

```
var v1 = 8;
console.log(v1.toString(2));//1000
var v2 = -8;
console.log(v2.toString(2));//-1000
```

抽 丝 剥 茧

toString(2)表示将数值转换成二进制，需要注意，这个方法转换后将无用的二进制位省略了，并且对于负数，其并没有转换成补码形式或者有符号数的二进制形式，而是采用了在其绝对值二进制形式的基础上加符号，这样一来，内存中数据存储时烦琐的转换过程对开发者来说都是透明的，开发者可以更轻松地理解自己所取到的数值。

下面将进入我们本节的正题：位运算。位运算顾名思义是在二进制位的基础上进行运算，其直接对二进制位进操作。JavaScript 中支持的位运算有 7 种，分别为按位非运算、按位与运算、按位或运算、按位异或运算、按位左移运算、按位有符号右移运算和按位无符号右移运算。

（1）按位非运算使用符号“~”来定义，也被称为按位取反运算，即原二进制位是 1 的变为 0，原二进制位是 0 的变为 1。示例代码如下：

```
var v3 = ~8;
console.log(v3);//-9
```

对 8 进行按位取反运算后，结果将为-9，如果将十进制换成二进制表示，这个过程就很好理解，首先 8 的二进制形式如下：

0	0	……	……	0	0	1	0	0	0

按位取反后如下：

1	1	……	……	1	1	0	1	1	1

前面说过，负数存储的实际上是补码，那么通过逆运算，先对补码减 1：

1	1	……	……	1	1	0	1	1	0

再对补码减 1 后得到的反码取反：

0	0	……	……	0	0	1	0	0	1

如上的原码就是我们最终结果的绝对值形式，将其转换成十进制并且加上负号，就得到了-9。

抽 丝 剥 茧

在编程中，你并不需要对每一次按位取反操作都进行如上推演，上面介绍的过程只是原理，理解了原理后，我们可以通过技巧记忆的方式来快速地得到想要的答案，对数值的按位取反操作实际上就是将此数值求负再减 1。

（2）按位与运算使用“&”符号定义，是一个二元运算符，其进行运算的两个操作数的对应二进制位分别进行与运算后将结果返回，即如果进行运算的相应位都为 1，那么最终结果数值的此二进制位为 1，否则为 0。示例代码如下：

```
var v4 = 1&9;
console.log(v4);//1
```

分解上述代码的计算过程如下：

① 1 的二进制码：

0	0	……	……	0	0	0	0	0	1

② 9 的二进制码：

0	0	……	……	0	0	1	0	0	1

③ 进行按位与运算后：

0	0	……	……	0	0	0	0	0	1

④ 最终结果为 1。

（3）按位或运算使用“|”符号定义，是一个二元运算符，其进行运算的两个操作数的对应二

进制位分别进行或运算后将结果返回，即如果进行运算的相应位都为 0，那么最终结果数值的二进制位为 0，否则为 1。示例代码如下：

```
var v5 = 8|3;
console.log(v5);//11
```

分解上述代码的计算过程如下：

① 8 的二进制码：

0	0	……	……	0	0	1	0	0	0

② 3 的二进制码：

0	0	……	……	0	0	0	0	1	1

③ 进行按位或运算后：

0	0	……	……	0	0	1	0	1	1

④ 最终结果为 11。

（4）按位异或运算使用符号“^”定义，是一个二元运算符，其进行运算的两个操作数的对应二进制位分别进行异或运算后将结果返回，即如果进行运算的相应位不同，那么最终结果数值的二进制位为 1，否则为 0。示例代码如下：

```
var v6 = 8^11;
console.log(v6);//3
```

分解上述代码的计算过程如下：

① 8 的二进制码：

0	0	……	……	0	0	1	0	0	0

② 11 的二进制码：

0	0	……	……	0	0	1	0	1	1

③ 进行按位异或运算后：

0	0	……	……	0	0	0	0	1	1

④ 最终结果为 3。

（5）按位左移运算使用符号“<<”定义，其作用是将二进制数据向左移动指定的位数，右侧空出来的位将进行补零操作，需要注意，按位左移操作并不会影响符号位，移动过程并不包括符号位，示例代码如下：

```
var v7 = -2<<2;
console.log(v7);//-8
```

分解上述代码的计算过程如下：

① -2 的二进制码（补码）：

1	1	……	……	1	1	1	1	1	0

② 进行左移两位的运算后：

1	1	……	……	1	1	1	0	0	0

③ 对补码求原码：

0	0	……	……	0	0	1	0	0	0

④ 最终结果为-8。

（6）与按位左移运算相对应的还有按位右移运算。需要注意，按位右移运算有两种：有符号按位右移运算与无符号按位右移运算。其中，有符号按位右移运算与按位左移运算互为逆运算，使用“>>”符号定义，示例如下：

```
var v8 = -8>>2;
console.log(v8);//-2
```

无符号按位右移运算和有符号按位右移运算最大的不同在于其在右移时，并不保留符号位，会将符号位一起进行移动，使用“>>>”符号定义。正数的符号位为 0，因此对正数并没有影响，负数就不同了，示例如下：

```
var v9 = 8>>>2;
console.log(v9);//2
var v10 = -8>>>2;
console.log(v10);//1073741822
```

具体过程，这里不再重复，可以根据前面的示例自行推导。最后，给一个中肯的建议，由于无符号右移运算的这种特性，在使用时要极其小心。

1.4.6 特殊运算符

C 语言中定义了自增与自减两种运算符，它们是很多初学者的噩梦。你或许猜到了，JavaScript 中也定义了这两个运算符并且和 C 语言中定义的用法基本一致。

自增运算符使用符号“++”定义，自减运算符使用符号“--”定义。简单理解，自增运算符是在操作数本身的基础上进行加 1 运算，自减运算符是在操作数本身的基础上进行减 1 运算，示例代码如下：

```
//自增与自减运算符
var a = 10;
var b = 10;
//进行自增与自减运算
a++;
b--;
console.log(a);//将打印 11
console.log(b);//将打印 9
```

需要注意，自增和自减运算符既可以放在操作数后面，也可以放在操作数前面。如果将运算符放在操作数后面，通常称其为“后置自增/减运算符”，如果将运算符放在操作数前面，通常称其为“前置自增/减运算符”。“前置”与“后置”虽然只是一字之差，其运算过程与结果却大相径庭。

先来看下面这个例子：

```
//自增/减运算符的前置与后置
var c = 10;
var d = 10;
console.log(c++);//将打印 10
console.log(++d);//将打印 11
console.log(c);  //将打印 11
console.log(d);  //将打印 11
```

单独打印变量 c 和变量 d 的结果都将是 11，说明无论是前置自增运算还是后置自增运算，其都是在原操作数的基础上进行加 1 运算。然而如果对“c++”和“++d”这两个表达式的返回值进行打印，可以发现前置自增运算返回的是运算完成后的值，而后置自增运算返回的是运算前的值，同样的规则也适用于自减运算符，示例代码如下：

```
var e = 10;
var f = 10;
console.log(e--);  //将打印 10
console.log(--f);  //将打印 9
console.log(e);  //将打印 9
console.log(f);  //将打印 9
```

在开发中，条件语句的编写必不可少，然而最简单的条件结构也需要至少 3 行功能代码，示例如下：

```
//条件结构
var res;
if(true){
    res = 10;
}else{
    res = 0;
}
console.log(res);//将打印 10
```

JavaScript 中提供了条件运算符“?:”来简化表达条件结构，上面示例可以简化成如下代码：

```
//条件表达式
var res = true?10:0;
console.log(res);//将打印 10
```

条件运算符组成的表达式结构为“逻辑值?表达式 1:表达式 2”，当问号前面的逻辑值为 true 时，运算结果为表达式 1 的值，当问号前面的逻辑值为 false 时，运算结果为表达式 2 的值。

JavaScript 中还定义了一种逗号运算符，其作用是将多个表达式放入一行语句中执行，示例如下：

```
//逗号表达式
var r1 = 1+3,r2=1*3;
console.log(r1),console.log(r2);//将打印 4 3
```

逗号运算符在进行多个变量的声明时十分方便。

JavaScript 中还提供了一个十分特殊的运算符“delete”，“delete”运算符用于将对象中的某个属性删除，示例如下：

```
var obj = {
    name:"珲少",
    age:25
};
console.log(obj.name);//将打印珲少
delete obj.name;
console.log(obj.name);//将打印 undefined
```

关于对象的更多内容，后面章节会详细介绍，这里你只需要了解“delete”运算符的作用即可。

1.4.7 运算符的优先级与结合性

在任何编程语言中，运算符的优先级与结合性都是一个老生常谈的话题。小学数学老师都一遍遍地告诉过我们“先乘除，后加减”的法则。在 JavaScript 语言中，也遵守类似的法则。例如，如下表达式计算的值是 22 而不是 28：

```
var res =  2+5*4;
console.log(res);//结果为 22
```

所谓运算符的优先级是指不同运算符在同一个表达式当中执行运算的先后顺序。优先级高的运算符将优先被执行，例如上面示例代码中“*”运算符的优先级要高于“+”运算符，因此先进行乘法运算再进行加法运算。

除了“优先级”的概念，运算符还有“结合性”概念。对于优先级相同的运算符，“结合性”决定了其表达式中运算的执行顺序。结合性分为左结合性和右结合性，左结合性的运算符将从左向右依次执行，右结合性的运算符将从右向左依次执行，示例如下：

```
//结合性
//左结合性
var a = 1+2+3;//结果为 6，相当于(1+2)+3
//右结合性
var b = c = 5;//相当于 c=5; b=c;
```

常用运算符的优先级与结合性见表 1-6。

表 1-6　运算符的优先级与结合性

运算符	优先级	结合性	运算符	优先级	结合性
小括号：()	19	-	按位无符号右移:>>>	12	左
后置递增：++	16	-	小于：<	11	左
后置递减：--	16	-	小于等于：<=	11	左
逻辑非：!	15	右	大于：>	11	左
按位非：~	15	右	大于等于：>=	11	左
正号运算符：+	15	右	等于：==	10	左
负号运算符：-	15	右	非等：!=	10	左
前置递增：++	15	右	全等：===	10	左
前置递减：--	15	右	非全等：!==	10	左
delete	15	右	按位与：&	9	左
乘法：*	14	左	按位异或：^	8	左
除法：/	14	左	按位或：|	7	左
取模：%	14	左	逻辑与：&&	6	左
加法：+	13	左	逻辑或：||	5	左
减法：-	13	左	条件：?:	4	右
按位左移：<<	12	左	赋值：=	3	右
按位右移：>>	12	左	逗号：,	0	左

博 闻 强 识

你能猜出下面代码的计算结果吗？

```
//例子
var i=3;
var j=3;
var n=3;
var a = i++ + i++ + i++; //3+4+5
var b = ++j + ++j + ++j; //4+5+6
var c = n++ + ++n + n++; //3+5+5
console.log(""+i+" "+j+" "+n+" "+a+" "+b+" "+c);//6,6,6,12,15,13
```

无论你对运算符的优先级与结合性记忆如何，都给出一个建议，如果有控制运算顺序的必要，请强制使用小括号，一目了然，省时省心。

第 2 章

JavaScript 流程控制与函数

流程控制是一门语言的基本功能。试想一下，有了流程控制，程序才有了自己的选择性与逻辑性。在 JavaScript 中提供的流程控制语句与 C 语言中支持的流程控制语句基本一致，但无疑 JavaScript 更加灵活。本章将向你介绍如何在 JavaScript 中使用条件语句、循环语句、中断语句等流程控制语句结构。

函数是功能独立的代码块，虽然和数学上函数的概念有一些区别，但都是用来解决某种问题，通俗地说，都是由一定的输入来获取运算后的输出（虽然在编程中的函数输入和输出都可能为空，有用的只是运算过程）。在 JavaScript 中，函数和其他数据类型一样也是一种对象，本章将向你简单地介绍函数的入门知识。

2.1　条件分支结构

城市里遍布着各种各样的十字路口，十字路口的贯通使得城市的交通四通八达。在程序的世界中也是一样，分支结构在任何语言中都是非常重要的一部分，如果没有分支结构，程序处理一个简单的选择逻辑都将变得十分困难。JavaScript 和大多数流行编程语言一样，定义了两种分支结构，分别为 if-else 分支结构和 switch-case 分支结构。

2.1.1　if-else 分支结构

if-else 语句是 JavaScript 中最常用的条件语句（大多数编程语言都是如此）。使用 if 语句，开发者可以根据不同的条件做不同的逻辑处理。最简单的 if 语句示例如下：

```
//if 结构 1
var condition = 10>5;
```

```
if (condition) {
    console.log("分支一");//将进行执行
}
console.log("结束");
```

if 关键字后面的小括号中需要写入一个要进行判断的条件，此表达式不一定是严格的 Boolean 类型，JavaScript 会自动对其进行 Boolean 类型转换。需要注意，为了减少歧义与出错率，建议在 if 条件中编写严格返回 Boolean 值的表达式。如果 if 条件最终为 true，则程序会执行大括号中的代码块；如果 if 条件最终为 false，则程序会跳过大括号中的代码块向后执行。

if-else 语句还有额外两种结构，示例如下：

```
//if 结构 2
if (condition) {
    console.log("分支一");
}else{
    console.log("分支二");
}
console.log("结束");
//if 结构 3
if (condition) {
    console.log("分支一");
}else if(condition2){
    console.log("分支二");
}else if(condition3){
    console.log("分支三");
}else{
    console.log("分支四");
}
console.log("结束");
```

结构 2 与结构 1 的区别在于：如果 if 条件为 false，则结构 1 会跳过 if 结构，结构 2 会执行 else 对应大括号的代码块。结构 3 是一种多条件分支结构，会依次判断 if 条件是否为 true，遇到一个 if 条件为 true 后则将执行对应的代码块，并且其后的 if 结构都会被跳过。

2.1.2　switch-case 分支结构

在学习 switch-case 结构之前，我们先来思考一个简单的场景：学生综合成绩满分 5 分，最低 1 分。分数从高到低依次代表卓越、优秀、良好、及格和不及格。试编写 JavaScript 程序来自动输出学生分数对应的档次。

使用 2.1.1 小节中学习的 if-else 结构，可以编写出如下代码：

```
var score = 4;
if (score==1) {
    console.log("不及格");
}else if(score==2){
```

```
        console.log("及格");
    }else if(score==3){
        console.log("良好");
    }else if(score==4){
        console.log("优秀");
    }else if(score==5){
        console.log("卓越");
    }else{
        console.log("无效的分数");
    }
```

上面的示例代码可以完成题目的要求，但是大量的 if-else 判断使代码变得十分冗余，看上去并不十分简洁，使用 switch-case 结构可以很好地解决这一问题。

switch-case 也是 JavaScript 语言中的一种多分支结构，使用 switch-case 结构重新改写上面的代码如下：

```
    switch (score) {
        case 1:{
                console.log("不及格");
            }
            break;
        case 2:{
                console.log("及格");
            }
            break;
        case 3:{
                console.log("良好");
            }
            break;
        case 4:{
                console.log("优秀");
            }
            break;
        case 5:{
                console.log("卓越");
            }
            break;
        default:{
                console.log("无效的分数");
            }
    }
```

switch 关键字后面需要指定一个表达式，case 子句对应的值如果和此表达式的值相等，则会执行此 case 对应的代码块。需要注意，break 语句用于跳出 switch-case 结构，即一旦某个 case 匹配成功，则不再进行后续 case 的匹配，这点在开发中要根据实际情况选择是否使用 break 来做跳出，default 块则是当所有 case 匹配都失败后会被执行的代码。

抽 丝 剥 茧

许多类 C 语言中的 switch-case 结构都有一个特殊的要求，其进行匹配的值必须为整型数据，JavaScript 则与其不同，switch-case 结构进行匹配的值可以是字符串甚至其他变量。

2.2　循环结构

循环结构又叫迭代结构，用来多次重复执行某一段逻辑代码。我们在计算数学题时，思路是向着简单化与公式化的，因为人的大脑有极限，无法也不可能进行超大量的计算工作。但是计算机解决问题的思路与之刚好相反，计算机的运算速度非常快，一般编程中根本无须考虑运算性能的问题。因此，作为开发者，在编写代码时更应该考虑的是代码的简洁与易读性。

2.2.1　while 循环结构

如果有人问你从 1 依次递增 100 的连加结果是多少。你可能首先会想到等差数列的求和公式：首项加末项的和乘以项数除以二。在编程中要解决这个问题，使用循环结构将更加简单。

下面的示例代码分别演示了使用数学公式的方式和使用循环结构的方式对等差数列进行求和运算：

```
//数学公式进行等差数列的计算 1…100
var res = (1+100)*100/2;
console.log(res);//5050
//使用循环结构来进行计算
var i=1;
var res2 = 0;
while(i<=100){
    res2+=i;
    i++;
}
console.log(res2);//5050
```

上面示例代码中使用到的 while 结构是 JavaScript 中十分常用的一种循环结构，while 关键字后面需要指定一个循环条件，当此条件成立时会执行 while 循环体中的代码，执行完成后，会继续进行循环条件是否成立的判断，如果条件成立会再次执行循环体直到条件不成立为止。

while 循环还有一种变体，其被称为 do-while 循环结构，与 while 循环的区别在于 do-while 结构需要先执行一次循环体，之后进行循环条件的判断，如果条件成立，会再次执行循环体直到条件不成立为止，上面的求和运算使用 do-while 结构进行改写：

```
//do-while 循环
i=1;
res3 = 0;
```

```
do{
    res3+=i;
    i++;
}while(i<=100);
console.log(res3);//5050
```

抽丝剥茧

在编程中的无限循环也叫做死循环。一般情况下，我们要避免编写出死循环的代码。使用 while 结构编写最简单的无限循环如下:

```
while(true){
    console.log("...");
 }
```

2.2.2 for 循环结构

for 循环结构是比 while 循环结构更加常用的一种循环结构，在许多其他流行编程语言中也是如此。for 循环的基本编写格式如下：

```
for(init;exp;exc){}
```

其中，init 语句对循环变量进行初始化，exp 语句是循环的条件，exc 语句用于修改循环变量，示例如下：

```
//for 循环
var res4 = 0;
for (var i = 1; i <= 100; i++) {
    res4+=i;
}
console.log(res4);//5050
```

for 循环语句的格式十分自由，init 语句、exp 语句甚至 exc 语句都是可以省略的。如果已经存在可以作为循环变量的变量，则 init 语句可以省略，示例如下：

```
var res4 = 0;
var i = 1;
for (; i <= 100; i++) {
    res4+=i;
}
console.log(res4);//5050
```

同样如果循环变量不需要修改，exc 语句也可以省略，示例如下：

```
//for 循环
var res4 = 0;
var i = 1;
for (; i <= 100;) {
```

```
        res4+=i++;
    }
    console.log(res4);//5050
```

需要注意，如果将循环条件进行了省略，则 for 循环会无限循环下去除非遇到中断语句。

抽 丝 剥 茧

用 for 循环结构编写最简单的无限循环，代码如下：

```
    for(;;)
```

JavaScript 中还提供了一种严格的迭代结构 for-in 枚举。for-in 枚举的作用是用来循环枚举出对象中所有可枚举的属性，示例代码如下：

```
for (prop in teacher) {
    /*
    将输出
    name:珲少
    age:25
    subject:JavaScript
    */
    console.log(prop + ":" + teacher[prop]);
}
```

2.3　中断与跳转结构

JavaScript 中的中断语句有两种，分别为 break 语句与 continue 语句。中断语句与标签语句结合使用，可以更加灵活地控制程序代码的执行流程。

2.3.1　break 语句

在讲解 switch-case 结构时我们已经使用过 break 语句，break 语句是 JavaScript 语言中的一种中断结构，在 switch-case 结构中，break 语句可以直接跳出当前的 switch-case 模块。同样，在循环结构中，也可以使用 break 语句来提前终止循环。示例代码如下：

```
//break 语句
for(var i=0;i<5;i++){
    console.log(i);//依次输出 0，1，2，3
    if (i==3) {
        break;
    }
}
```

在上面的代码中，如果不添加 break 语句，程序将依次输出“0，1，2，3，4”。添加了 break 语句后，当循环变量 i 自增到 3 时，程序将跳出循环结构，控制台只会输出“0，1，2，3”。需要注意，break 语句默认将跳出当前所在的最内层循环。下面的代码演示如何在多层循环中使用 break 语句：

```
for(var i=0;i<3;i++){
    console.log("i="+i);
    for(var j=0;j<3;j++){
        if (i==0) {
            break;
        }
        console.log("j="+j);
    }
    console.log("=====");
}
/*
打印结果
i=0
=====
i=1
j=0
j=1
j=2
=====
i=2
j=0
j=1
j=2
=====
*/
```

从程序的打印结果可以看出，break 语句跳出了内层循环，外层循环依然继续执行。

对于多层嵌套的循环，如果需要灵活控制所跳出的循环，可以将 break 语句与标签语句结合使用，示例如下：

```
label:for(var i=0;i<3;i++){
    console.log("i="+i);
    for(var j=0;j<3;j++){
        if (i==0) {
            break label;
        }
        console.log("j="+j);
    }
    console.log("=====");
}
```

```
/*
打印结果
i=0
*/
```

从打印结果可以看出，上面代码中的 break 语句直接跳出了外层循环。标签语句的格式十分简单，在要加标签的语句前使用标签名加冒号的形式标注即可，break 关键字后面可以指定一个已存在的标签名来跳出特定的循环。

博 闻 强 识

从原理上讲循环结构是可以无限嵌套的，但是在实际开发中，代码的整洁与易读性也十分重要，编写循环结构时，应尽量保持不超过 3 层嵌套。中断语句与标签语句的结合使用可以极大地提高程序的灵活性，同时也增加了代码的调试难度。如果不是非常需要，尽量少使用标签语句，如果一定要用，请让标签的命名更易于理解。

2.3.2　continue 语句

continue 语句也是 JavaScript 中常用的中断语句，其和 break 语句有很大的区别。break 语句是跳出当前的循环结构，而 continue 语句则是跳出本次循环。示例如下：

```
//continue 语句
for(var i=0;i<5;i++){
    if (i==2) {
        continue;
    }
    console.log(i);//依次输出 0，1，3,4
}
```

从结果可以看出，continue 语句的作用是使程序仅仅跳过了循环变量 i 等于 2 时的输出代码，并不会影响循环的再次执行。同样，对于多层循环结构，continue 语句默认也将跳出当前循环的本次循环，示例如下：

```
for(var i=0;i<3;i++){
    console.log("i="+i);
    for(var j=0;j<3;j++){
        if (j==0) {
            continue;
        }
        console.log("j="+j);
    }
    console.log("=====");
}
/*
打印结果
```

```
i=0
j=1
j=2
=====
i=1
j=1
j=2
=====
i=2
j=1
j=2
=====
*/
```

continue 语句也可以和标签语句结合使用来跳出指定循环结构的本次循环，示例如下：

```
label2:for(var i=0;i<3;i++){
    console.log("i="+i);
    for(var j=0;j<3;j++){
        if (j==0) {
            continue label2;
        }
        console.log("j="+j);
    }
    console.log("=====");
}
/*
打印结果
i=0
i=1
i=2
*/
```

需要注意，上面的代码没有打印出任何 j 值的原因是当判断到 j 等于 0 时，直接跳出了外层循环的本次循环，因此内层循环的循环变量 j 并没有执行递增操作，j 始终为 0。

2.4 异常捕获结构

无论你是经验多么丰富的开发者，在运行所编写的代码时，一定会发生错误。在 JavaScript 引擎执行 JavaScript 代码时，会发生各种各样的错误，可能是语法错误，也可能是数据处理错误，还可能是由于服务端数据或者用户输入数据超出开发者意料发生的逻辑错误，等等。JavaScript 内置了许多异常对象，当某些行为触发了这些异常时，程序会将异常抛出并且中断在抛出异常的地方。

开发者可以使用 try-catch 结构捕获并处理异常来保证程序的顺畅执行，同样开发者也可以使用 throw 语句来抛出一个异常。

2.4.1　使用 throw 语句抛出异常

暴露错误和解决错误同样重要。在开发中你或许会编写大量的有参数输入的函数，对于实际输入函数的参数，有时候并不是你可以决定的，例如我们要编写一个标准的除法程序，代码如下：

```
function div(a,b){
    return a/b;
}
var res = div(3,4);
console.log(res);
```

上面的代码乍看起来很完美，实际上问题很多，对于一个标准的除法运算，首先其除数与被除数应该都是标准的数值类型，不可以是字符串或者其他对象。其次，除数与被除数都不可以为无穷值，被除数也不可以为 0。上面这两条规则都是在对传入的参数进行限制，你或许会想，我们并没有方法控制用户要输入的东西啊，没错，但是你可以在用户输入了错误数据时让函数抛出异常，使程序中断。

JavaScript 中使用 throw 关键字来进行异常的抛出，修改上面的代码如下：

```
function div(a,b){
    if ((typeof a) != "number"||(typeof b) != "number"){
        throw "must input a Number Value";
    }
    if (!isFinite(a)|| !isFinite(b)) {
        throw "must input a Number is Finity";
    }
    if (b===0) {
        throw "Dividend must not be 0";
    }
    return a/b;
}
var res = div(3,4);
console.log(res);
```

当输入非数值类型的参数、数值为无穷的参数或者被除数为 0 时，程序会直接中断，并在控制台打印抛出的异常信息，如此便十分严格地对函数的输入参数进行了约束。

使用 throw 关键字进行异常抛出的基本语法结构为 throw exp。其中，throw 为系统关键字，exp 是要抛出的异常表达式，其可以为任意类型。例如，上面示例代码中我们实际上抛出的是一个字符串值异常，也可以抛出数值、布尔值或者任意自定义对象。例如，我们可以为某种特定的错误类型创建一个专门定义的对象，示例如下：

```
var InputError = {
    NotNumberError : "must input a Number Value",
```

```
    FinityError : "must input a Number is Finity",
    DividendError:"Dividend must not be 0"
};
function div(a,b){
    if ((typeof a) != "number" || (typeof b) != "number"){
        throw InputError.NotNumberError;
    }
    if (!isFinite(a)|| !isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b===0) {
        throw InputError.DividendError;
    }
    return a/b;
}
var res = div(3,4);
console.log(res);
```

通过自定义异常对象的方式抛出异常有利于更结构化和标准化地对异常进行捕获与处理。现在，你已经学会了如何在代码中抛出异常，但仅仅抛出异常是远远不够的，后面我们将学习如何捕获异常、处理异常与传递异常。

2.4.2 异常的捕获与处理

JavaScript 代码中抛出的异常如果不进行处理程序会直接中断，这是我们不想看到的结果。JavaScript 中的 try-catch-finally 结构用来捕获和处理异常。依然使用 2.4.1 小节所编写的除法器来做例子，在调用除法器函数时，传入一个错误的参数：

```
var InputError = {
    NotNumberError : "must input a Number Value",
    FinityError : "must input a Number is Finity",
    DividendError:"Dividend must not be 0"
};
function div(a,b){
    if ((typeof a) != "number" || (typeof b) != "number"){
        throw InputError.NotNumberError;
    }
    if (!isFinite(a)||!isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b===0) {
        throw InputError.DividendError;
    }
    return a/b;
}
```

```
var res = div("3",4);
console.log(res);
```

不出所料，程序运行时会抛出异常并直接中断。在实际开发中，遇到异常后让程序中断并不是一个友好的选择（尽管用户可能输入的数据有些随心所欲），我们应该想办法将错误的信息提示给用户，并且让程序跳过这个异常。很多情况下，一个复杂的程序不只有一个功能，因为某个功能的异常而退出整个程序是让人无法接受的。try-catch-finally 结构在 JavaScript 中专门用来捕获与处理异常。

博 闻 强 识

很多高级语言都有异常处理机制，有编程基础的同学一定对 try-catch-finally 十分熟悉，在 Java 语言中也有这样的结构。Objective-C 语言中对应的结构为@try-@catch-@finally。Swift 语言中的异常处理略微复杂一些，但是也有 do-catch 结构。

将上面的示例程序修改如下：

```
var InputError = {
    NotNumberError : "must input a Number Value",
    FinityError : "must input a Number is Finity",
    DividendError:"Dividend must not be 0"
};
function div(a,b){
    if ((typeof a) != "number" || (typeof b) != "number"){
        throw InputError.NotNumberError;
    }
    if (!isFinite(a)||!isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b===0) {
        throw InputError.DividendError;
    }
    return a/b;
}
try{
    var res = div("3",4);
}catch(e){
    console.log(e);
}finally{
    console.log("异常处理结束");
}
console.log(res);//将打印 undefined
```

运行代码，控制台的打印结果如图 2-1 所示。

```
must input a Number Value
异常处理结束
undefined
[Finished in 0.1s]
```

图 2-1 进行异常捕获

可以发现，这次程序完整运行，下面我们来解释控制台的打印信息是如何来的。首先，在 try-catch-finally 结构中的 3 个关键字里，try 对应的代码块用来执行可能会抛出异常的代码，实际上，如果没有异常抛出，catch 结构就是透明的，如果 try 关键字对应的代码块中有异常抛出，程序首先会进入 catch 代码块，catch 代码块的作用是对抛出的异常进行捕获，其会将抛出的异常作为参数传入，我们拿到异常后，可以根据异常类型做相关逻辑处理。异常处理结束后，如果有 finally 代码块，就会执行 finally 代码块中的代码，之后结束异常捕获处理结构。在 try-catch-finally 结构中，不一定 3 个代码块都要存在，可以有如下 3 种结构：

- try-catch
- try-finally
- try-catch-finally

需要注意，finally 代码块并不对异常进行捕获，无论有没有异常抛出，finally 代码块都会被执行，同样如果没有 catch 代码块但是抛出了异常，尽管 finally 代码块存在，程序依然会中断。

2.4.3 异常的传递

前面你已经学会了如何捕获与处理异常，下面我们将除法器函数再进行一层封装，将异常处理逻辑封装进新的函数内部，这样在其他开发者调用这个新的除法器函数时就无须再担心程序的中断了，示例代码如下：

```
var InputError = {
    NotNumberError: "must input a Number Value",
    FinityError: "must input a Number is Finity",
    DividendError: "Dividend must not be 0"
};
function div(a, b) {
    if ((typeof a) != "number" || (typeof b) != "number") {
        throw InputError.NotNumberError;
    }
    if (!isFinite(a) || !isFinite(b)) {
        throw InputError.FinityError;
    }
    if (b === 0) {
        throw InputError.DividendError;
    }
    return a / b;
}
function newDiv(a,b) {
```

```
        try {
            var res = div(a, b);
        } catch (e) {
            console.log(e);
        } finally {
            console.log("异常处理结束");
        }
        return res;
    }
    var res = newDiv("3",4);
    console.log(res);
```

经过优化后，在调用上面代码中的 newDiv()函数时，无论传入的参数怎样都已经不会再抛出异常，异常已经在 newDiv()函数内部被捕获并且处理了。如果我们不在 newDiv()函数内部对异常进行捕获处理，异常会被继续向上传递，从 newDiv()函数抛出，示例如下：

```
    function newDiv(a,b) {
        try {
            var res = div(a, b);
        } finally {
            console.log("异常处理结束");
        }
        return res;
    }
    try{
        var res = newDiv("3",4);
    }catch(e){
        console.log("newDiv 处理异常");
    }
```

控制台打印信息如图 2-2 所示。



图 2-2　异常的传递

在 JavaScript 中，一旦某个函数抛出异常，异常首先会传递到此函数的调用方，如果调用方有对异常进行捕获处理，则异常不再向上传递，如果调用方没有对抛出的异常进行捕获处理，则此异常会继续传递到调用方的调用方，按照这样的逻辑一层一层地向上传递，直到被捕获处理或者不再有可以处理的调用方。

明白了异常的传递机制，我们可以更加灵活地控制异常的传递，将异常进行分层处理，示例代码如下：

```
function newDiv(a,b) {
    try {
        var res = div(a, b);
    } catch(e){
        if (e===InputError.DividendError) {
            res = NaN;
        }else if (e==InputError.FinityError) {
            res = NaN;
        }else{
            throw e;
        }
    }finally {
        console.log("内层异常处理结束");
    }
    return res;
}
try{
    var res = newDiv(Infinity,1);
}catch(e){
    console.log("输入错误!");
}finally{
    console.log("外层异常处理结束");
}
console.log(res);
```

在上面的代码中，newDiv()函数只对被除数为零和参数有无穷值的异常进行了处理，将 NaN 直接作为结果返回，如果是参数类型错误，则 newDiv()函数不做任何判断，直接将异常抛出，交给调用方处理。

2.5 JavaScript 中的函数

几乎所有编程语言中都有函数这个概念，从功能上讲，函数是一组可以随时运行的代码语句，函数是一个代码块也是一段子程序。JavaScript 是一种面向对象的语言，在 JavaScript 的世界中，万事万物都可以是对象。有趣的是，函数在 JavaScript 中实质上也是一种功能完整的对象。JavaScript 中有 3 种基本的方式来定义函数，分别为函数语句定义法、函数表达式定义法和 Function 构造函数对象法。

2.5.1 使用函数语句定义函数

JavaScript 中有一种特殊的语法可以直接定义函数，在前面的讲解中，我们也经常使用这种方法来定义函数，示例如下：

```
function outputName(){
    console.log("Jaki");
}
```

函数语句的语法结构可以简化如下：function name(param…){}。其中，function 为定义函数的关键字，name 为函数的名称，通过函数名称可以直接调用函数，后面的小括号中可以定义函数的形参，大括号中为函数核心函数体，其中可以编写函数的具体逻辑代码。

博 闻 强 识

形参的全称为“形式参数”，其在定义函数的时候被定义，用来接收传入函数的参数。形参在整个函数内部使用，脱离函数内部，形参将没有任何意义。与形参对应的还有实参，实参的全称为“实际参数”，其是函数在调用时确确实实传递给函数的参数。一般情况下，在调用函数的时候，实参传递给形参并且一一对应，但是在 JavaScript 语言允许开发者传入的实参与形参并不对应。

下面我们来定义一个带参数和返回值的函数，代码如下：

```
function outputHello(name){
    return "Hello "+name;
}
console.log(outputHello("Lcuy"));//打印出 Hello Lcuy
```

上面的函数需要传入一个姓名作为参数，将姓名拼接上“Hello”字符串后返回。在 JavaScript 中，函数的返回值不需要专门定义（和其他主流编程语言有所区别），直接使用 return 关键字进行返回即可。

抽 丝 剥 茧

事实上，在 JavaScript 中任何一个函数都有返回值，如果不显式地使用 return 关键字进行返回，或者只使用了 return 关键字，而没有返回任何值，函数实际的返回值为 undefined。

JavaScript 语言是通过函数语句语法定义的函数，函数的调用和函数的定义并没有严格的顺序，其实我们也可以先调用某个函数，之后再进行此函数的定义，示例代码如下：

```
//先调用
console.log(outputHello("Lcuy"));//打印出 Hello Lcuy
//后定义
function outputHello(name){
    return "Hello "+name;
}
```

上面的代码可以顺利执行的原因很简单，JavaScript 语言中有“变量提升”这样一种概念。在程序执行之前，JavaScript 解释器会将全局声明的变量、定义的函数等进行预处理，因此在代码层面，函数的调用是在定义前还是定义后都无关紧要。

2.5.2 使用函数表达式定义函数

前面我们讲过，函数是一种功能完整的对象，在 JavaScript 中也可以使用函数表达式来定义函数对象。函数表达式看上去与函数语句十分相似，示例代码如下：

```
//函数表达式
var outputAge = function(age){
    console.log("My age is "+age);
};
outputAge(25);//将打印 My age is 25
```

上面的代码在定义函数时，等号左边是一个变量，等号右边是一个函数表达式。函数表达式实质上返回了一个函数对象，将其赋值给 outputAge 变量，之后通过 outputAge 变量便可以直接对其所指向的函数进行调用。需要注意，函数表达式与函数语句的最大语法区别在于其可以省略函数名，也可以理解为函数表达式创建了一个匿名函数。

函数表达式也可以创建有名称的函数，这在编写递归函数时十分重要，其方便了函数进行自调用，示例如下：

```
//定义一个递归阶乘函数
var mathFunc = function mathF(a){
    var res = a;
    a--;
    if (a>0) {
        res *= mathFunc(a);
    }
    return res;
};
var mathRes = mathFunc(5);
console.log(mathRes);//120
```

需要注意，函数表达式定义的函数名只能在函数内部使用，对外部来说，其实质上还是一种匿名函数。

还有一点需要特别注意，函数表达式定义的函数在函数定义之前是不能够进行调用的。你或许会觉得奇怪，JavaScript 解释器不是会将一些全局定义预处理吗？没错，只是函数表达式的实质是将一个函数对象赋值给了一个变量，JavaScript 解释器只是预先定义好了变量符号，在表达式未执行之前，此变量实际是 undefined，函数并不存在。这也是函数表达式与函数语句的另一重大区别，函数语句定义后的函数名是无法修改的，而函数表达式定义的函数只是将函数对象赋值给了变量，变量可以重新赋值，也可以将函数传递给另一个变量，示例如下：

```
//函数表达式
var outputAge = function(age){
    console.log("My age is "+age);
};
//将 outputAge 函数传递给新的变量
```

```
var newFunc = outputAge;
//重新对 outputAge 赋值
outputAge = "Hello world";
newFunc(25);//将打印 My age is 25
console.log(outputAge);//将打印 Hello world
```

2.5.3　使用 Function 构造函数

函数在 JavaScript 中是功能完整的对象，关于对象你现在了解的可能并不深入，不用担心，我们后面会有专门的章节介绍 JavaScript 中的对象。现在你只需要简单知道对象是属性和方法的包装即可。JavaScript 中也可以通过 Function 对象和 new 关键字来构造函数对象，简单代码示例如下：

```
var output = new Function("name","console.log(name);");
output("Jaki");//将打印输出 Jaki
```

Function 构造函数的结构可以简化为 Function(param,param…,funcbody)。Function 构造函数中的参数个数并不固定，最后一个参数为创建函数的函数体，前面所有的参数都将作为函数的形参。需要注意，所有的参数类型都需要为字符串类型，函数体也可包装成字符串。

事实上，无论通过哪种方式创建的函数实质上都是 Function 类型的对象。Function 实例对象中有一些内置的属性，常用的有 arguments 属性与 length 属性。arguments 属性将返回一个数组结构数据，数组中为开发者传入的所有实参。length 属性将返回函数形参的个数。有了 arguments 属性，我们在使用函数的时候就不必从形参获取外界传递进函数内部的数据了，示例如下：

```
function myFunc(){
    //将传入的参数倒叙输出
    for (var i = arguments.length - 1; i >= 0; i--) {
        console.log(arguments[i]);
    }
}
myFunc(1,2,3);//将输出 3, 2, 1
```

抽 丝 剥 茧
需要注意，函数对象的 arguments 属性内部也有一个 length 属性，它是数组对象内置的属性，用于获取数组中元素的个数，并不是函数对象的 length 属性。

函数对象是可以直接被调用执行的，无论是使用 Function 构造函数还是使用函数表达式定义函数，实际上都是创建了一个函数对象，将其赋值给了一个变量。对于一些只需要在创建时执行一次就不再需要的函数，也可以使用如下方式编写：

```
(function(){
    console.log("run self");
})();//将打印 run self
```

上面这种函数的应用方式在实际开发中应用十分广泛，可以用来包装作用域，隐藏某些内部变量和方法。

第 3 章

JavaScript 对象基础

向 Java、C++、Swift 编译型语言一样，JavaScript 也是一种面向对象的语言。在面向对象的语言世界里，大多数都是基于类的（可以想象，必须先有类才能构造实例），但 JavaScript 另辟蹊径，其抛弃了类的概念，通过原型来实现类的功能，也就是说，JavaScript 是一种基于原型的面向对象语言。在 JavaScript 中，除了几种原始类型外，万事万物都是对象。

3.1　初识 JavaScript 对象

“对象”一词相信你并不陌生，前面的章节中我们也有意无意地多次提到过对象这个概念。在程序的世界里，对象用来描述某个特定的事物，是现实世界中事物的抽象。面向对象是一种软件开发方式，在程序中使用对象技术来描述现实中的事物，可以使程序代码更易理解维护，同时抽象性、封装性和可重用性都会大大提高。

博 闻 强 识

从描述问题的方式来看，编程语言可以简单地分为面向过程语言和面向对象语言。面向过程语言以“数据结构+算法”的模式来解决问题。面向对象语言使用“对象+消息”的模式来解决问题。相比之下，面向过程语言编写的程序更简洁、更快，适合进行科学计算。面向对象编写的程序更抽象、更易懂，适合解决现实生活中的问题。

3.1.1　在 JavaScript 中创建对象

JavaScript 是一种基于对象的脚本语言，不仅可以使用语言内置的许多对象，还可以创建自己

所需要的对象。使用字面量创建一个对象的语法非常简单，只需要使用大括号将需要封装的属性和方法进行包裹即可，示例如下：

```
var teacher = {};
console.log(typeof teacher);//将打印 object
```

上面的第一句代码创建了一个空对象，将其赋值给了变量 teacher。如果使用 typeof 运算符对 teacher 变量的类型进行检查，你会发现它实际上是 object 类型，即对象类型。一般我们创建对象，是为了描述某个事物。以 teacher 对象为例，我们想让它描述一个教师的个人信息和行为，一个空的对象是没有实际用途的，为 teacher 对象添加一些属性和行为，代码如下：

```
var teacher = {
    name:"珲少",
    age:25,
    subject:"JavaScript",
    teaching:function(){
        console.log("开始教学");
    },
    relaxing:function(){
        console.log("开始讲故事");
    }
};
```

上面的代码为 teacher 对象添加了 3 个属性和 2 个行为。name 属性用来描述教师的姓名，age 属性用来描述教师的年龄，subject 属性用来描述教师的教学科目。teaching 行为用来描述教师的教学动作，relaxing 行为用来描述教师的休息动作。你应该明白了，属性描述的都是一些对象信息，其语法很像键值对，冒号左边是属性名，冒号右边是属性值。行为描述的都是对象动作，在语法上冒号左边是行为名称，冒号右边是行为函数。

抽 丝 剥 茧
对象属性的值可以是任何类型，当然也可以是另一个对象。

使用“点语法”可以方便地访问对象的属性和调用对象的方法。示例如下：

```
console.log(teacher.name);//打印教师的姓名
console.log(teacher.age);//打印教师的年龄
console.log(teacher.subject);//打印教师的专业
teacher.teaching();//执行教学动作
teacher.relaxing();//执行休息动作
```

除了点语法，JavaScript 中还可以通过中括号法来访问对象属性与调用对象的方法，示例如下：

```
console.log(teacher["name"]);//打印教师的姓名
console.log(teacher["age"]);//打印教师的年龄
console.log(teacher["subject"]);//打印教师的专业
teacher["teaching"]();//执行教学动作
teacher["relaxing"]();//执行休息动作
```

需要注意，使用中括号法访问对象的时候，中括号中的值必须为字符串类型。

3.1.2 设置对象的属性和行为

3.1.1 小节中在创建 teacher 对象时就已经为这个对象添加了一些属性和方法。很多时候对象并不是一成不变的，现实生活中也是这样，比如几个月前笔者还在教 Swift 编程语言，现在已经在和你交流 JavaScript 了。其实，我们可以随时为已经存在的对象添加新的属性方法或者修改某个属性与方法。为对象追加或修改属性与方法的语法也十分简单，示例如下：

```
//修改对象
teacher.subject = "Swift";//将专业修改为 Swift
teacher.teaching = function(){
    console.log("Teaching Swift");
};
teacher.students = ["July","Jeke","Even"];//添加一个学员列表
console.log(teacher);
```

同样，使用方括号法也可以完成对象的设置，示例如下：

```
//修改对象
teacher["subject"] = "Swift";//将专业修改为 Swift
teacher["teaching"] = function(){
    console.log("Teaching Swift");
};
teacher["students"] = ["July","Jeke","Even"];//添加一个学员列表
console.log(teacher);
```

抽 丝 剥 茧

你一定还记得，我们在学习运算符的时候提到 delete 运算符，它的作用就是删除对象的某个属性或行为。

在阅读 JavaScript 代码时，你可能会发现一个出现频率非常高的关键字：this。在对象内部行为使用的 this 关键字将指向当前对象本身（这是一般情况，并不是必然情况，this 的指向会随着运行环境或调用者而灵活修改）。例如：

```
var person = {
    name:"Jaki",
    sayHi:function(){
        console.log("Hi,My name is " + this.name);
    }
};
person.sayHi();//将输出 Hi,My name is Jaki
```

上面的代码中 this 指的就是 person 对象本身，其实上面代码和下面代码的功能完全一致：

```
var person = {
    name:"Jaki",
    sayHi:function(){
        console.log("Hi,My name is " + person.name);
    }
};
```

你可能会问，那我们直接使用 person 不就好了？为什么非要使用 this 呢？针对上面的情况，的确如此，我们没有必要使用 this 关键字。等到后面我们更加深入地学习了面向对象编程，尤其是动态生成对象的方法后，你就能真正体会到 this 关键字的强大之处了，现在让我们循序渐进，拭目以待！

3.2　JavaScript 中常用的内置对象

在 JavaScript 中有 5 种原始类型（Undefined、Null、Boolean、Number、String），针对 Boolean、Number 和 String 原始类型，在 JavaScript 中还有其对象形式的包装，并且在必要时，JavaScript 会自动地在原始值和对象之间转换。由于这些内置对象的存在，我们可以方便地对数值、字符串等数据进行操作和处理。所谓内置对象，是针对自建对象而言的，在 3.1 节中我们创建的“教师”对象就是自建对象。内置对象则是 JavaScript 中预先定义为开发者提供的对象，开发者可以直接使用。

3.2.1　JavaScript 中的 Number 对象

JavaScript 中的 Number 对象是对原始类型 Number 的包装。可以使用 Number 构造函数来进行实例创建，其中需要传入被创建对象的数字值。示例如下：

```
var num1 = new Number(5);
console.log(num1);//将打印[Number: 5]
console.log(typeof num1);//将打印 object
```

如果传入 Number 方法中的参数无法转换成数字，则会返回 NaN。

在 JavaScript 中，构造函数也是一种对象。实际上，Number 函数本身也是一个对象（我们先将其称为 Number 函数对象），只是它的作用是生成其他对象（姑且把它生成的对象都叫做 Number 实例对象），JavaScript 通过 new 关键字生成对象的原理这里先不做过多介绍，后面会专门讨论 JavaScript 中的原型机制。现在，你只需要区别开 Number 函数对象与 Number 实例对象即可。在 Number 函数对象中定义了许多有意义的常量属性，示例如下：

```
//可以表示的两个数值之间的最小间隔
console.log(Number.EPSILON);//2.220446049250313e-16
//JavaScript 中最大的安全整数
console.log(Number.MAX_SAFE_INTEGER);//9007199254740991
//能表示的最大数
```

```
console.log(Number.MAX_VALUE);//1.7976931348623157e+308
//能表示的最接近 0 的数
console.log(Number.MIN_VALUE);//5e-324
//非数字值
console.log(Number.NaN);//NaN
//负无穷大
console.log(Number.NEGATIVE_INFINITY);//-Infinity
//正无穷大
console.log(Number.POSITIVE_INFINITY);//Infinity
```

正负无穷值当产生溢出行为时会被返回。Number 函数对象中也定义了一些常用的方法，示例如下：

```
//判断传入的参数是否是 NaN
console.log(Number.isNaN(1));
//判断是否是有限数字
console.log(Number.isFinite(1));
//判断是否为整数，字符串将输出为 false
console.log(Number.isInteger("1"));
//判断是否为安全的整数
console.log(Number.isSafeInteger(1));
//将字符串转换为浮点值
console.log(Number.parseFloat("1.23"));
//将字符串转换为整数值
console.log(Number.parseInt("123.12"));
```

上面的属性和方法都是 Number 函数对象定义的，Number 实例对象中也有一些预定义的方法，所有的 Number 实例对象共享这些方法，示例如下：

```
var num2 = new Number(123);
//将数字转换成科学计数法 传入的参数为保留小数的位数
console.log(num2.toExponential(2));//1.23e+2
//将数字转换成字符串 传入的参数为保留小数的位数
console.log(num2.toFixed(2));//123.00
//将数字转换为指定有效数字长度的数字 传入的参数为有效数字的位数
console.log(num2.toPrecision(2));//1.2+e2
//将数字转换成字符串 传入的参数设置进制
console.log(num2.toString(10));//123
//返回 Number 实例对象的原始值
console.log(num2.valueOf());
```

抽 丝 剥 茧

在使用 toString()函数将数值对象转换为字符串时，可以传入参数来设置要转换成的进制。这个进制参数可以设置为 2～36 的一个整数，这个范围很好理解，数字 0～9 加上 26 个英文字母，最多可以表达 36 个数字，因此最大可以描述到三十六进制。

需要注意，如果使用字面量创建了一个原始数值，实际上也可以调用上面 Number 实例对象的方法，我们讲过，JavaScript 会在必要的时候进行原始值与对象的转换，示例如下：

```
var num3 = 100;
console.log(typeof num3);//number
var str = num3.toString();
console.log(typeof str);//string
console.log(typeof num3);//number
```

注意，num3 在调用 toString()方法的时候，JavaScript 把其当作对象来处理，但是并没有真正将其转换成 Number 实例对象。JavaScript 的这种弱类型的设计风格使我们在编写代码时十分畅快。

博 闻 强 识

使用 new 关键字在构造对象时实际上执行了 3 个操作，首先创建一个空的对象建立原型链，之后执行构造函数，将函数中的 this 关键字与新建的对象进行绑定，最后将这个新建的对象返回。因此，new 关键字最大的作用不仅仅是创建出一个新的对象，而是原型链的建立。在很多场景中，你可能会见到直接使用 Number 函数来创建数值，并不使用 new 关键字。需要注意，这种方式创建的是原始数值，并不是 Number 实例对象，通常我们可以使用这种方式来完成其他类型到数值类型的强转，示例如下：

```
var num4 = Number(5);
console.log(typeof num4);//number
```

做一点点小扩展，严格地讲，JavaScript 中是没有“类”这样一个概念的，这与许多面向对象语言不同。以 Swift 语言为例，对象的创建都是基于类的，我们会将一些所有对象共享的数据定义为静态属性或静态方法，每个对象独有的数据定义为成员属性或成员方法，并且类可以通过继承来派生出子类。在 JavaScript 中，我们也可以将有构造函数性质的函数对象理解为“类”，这个函数对象包含的属性和方法就类似于 Swift 语言中类的静态属性和静态方法，这个函数构造出来的对象的属性方法就类似于 Swift 语言中类的成员属性和成员方法，并且通过原型链 JavaScript 中也可以实现“类”的继承，后面我们会做深入介绍。

3.2.2　JavaScript 中的 String 对象

String 对象是对字符串的一种包装。在 JavaScript 中可以使用双引号或者单引号来创建字符串原始值，同样可以使用 String()构造函数来创建字符串对象，示例如下：

```
var str1 = new String("Hello World");
console.log(str1);//[String: 'Hello World']
```

也可以不使用 new 关键字直接使用 String()函数，只是这样的话实际上是创建了一个原始类型的 String 字符串，并不是对象，这种方法常常用来进行字符串转换。在实际开发中，对字符串的操作将十分频繁，举例来说，进行 URL 协议解析时，你可能需要截取出 URL 中定义的参数。在界面数据的填充时，你可能需要对字符串进行拼接、插入或替换等操作。JavaScript 中的 String 对象中封装了许多属性和方法，可以帮助开发者便捷地对字符串进行操作。

任何 String 实例对象中都包含一个 length 属性，通过 length 属性可以获取到字符串长度，即字符个数，示例如下：

```
var str1 = new String("Hello World");
console.log(str1.length);//11
```

JavaScript 最初的设计是用于编写浏览器脚本，因此其 String 实例对象中内置了许多对 HTML 标签操作的方法，这不是本书的重点，这里我们就不再进行深入讨论，你只需要掌握字符串操作相关的方法即可，示例如下：

```
var str1 = new String("Hello World");
//返回特定位置的字符，下标从 0 开始
console.log(str1.charAt(0));//H
//返回特定位置的字符编码值
console.log(str1.charCodeAt(0));//72
//在字符串后进行拼接，将拼接后的字符串返回
console.log(str1.concat("!"));//Hello World!
//获取某个字符在字符串中的索引，从前往后找，如果没有找到将返回-1
console.log(str1.indexOf('l'));//2
//获取某个字符在字符串中的索引，从后往前找，如果没有找到将返回-1
console.log(str1.lastIndexOf('l'));//9
//进行字符串的比较，原字符串小于参数字符串则返回小于 0 的数，大于则返回大于 0 的数，相等则
返回 0
console.log(str1.localeCompare("Aello World"));//1
//使用正则表达式对字符串进行匹配，匹配结果将返回一个对象
console.log(str1.match(/He/));//[ 'He', index: 0, input: 'Hello World' ]
//使用正则表达式来匹配字符串，将匹配到的字符串进行替换
console.log(str1.replace(/He/,"AI"));//AIllo World
//使用正则表达式来查找某个子串的位置，如果没有找到，则返回-1
console.log(str1.search(/He/));//0
//截取范围内的子字符串
console.log(str1.slice(0,3));//Hel
//分隔字符串，返回数组，其中第 1 个参数为进行分割的字符，第 2 个参数为返回最多子串个数
console.log(str1.split("l",10));//[ 'He', '', 'o Wor', 'd' ]
//进行字符串的截取，第 1 个参数为开始截取的位置，第 2 个参数为截取的长度
console.log(str1.substr(0,2));//He
//截取下标间的子串
console.log(str1.substring(1,2));//e
//将字符串转为小写
console.log(str1.toLocaleLowerCase());//hello world
console.log(str1.toLowerCase());//hello world
//将字符串转换为大写
console.log(str1.toLocaleUpperCase());//HELLO WORLD
console.log(str1.toUpperCase());//HELLO WORLD
//去掉字符串开头和结尾的空格
console.log(str1.trim());
```

```
//从字符串左侧去掉空格
console.log(str1.trimLeft());
//从字符串右侧去掉空格
console.log(str1.trimRight());
//获取字符串对象的原始值
console.log(str1.valueOf());
```

注意，上面的所有方法并没有修改原字符串对象，而是将结果以字符串原始值的形式进行返回。和大多数编程语言一样，JavaScript 中字符串的下标也是从 0 开始。在上面的示例代码中，match、replace 和 search 方法都是通过正则表达式来进行子串的匹配，正则表达式用来描述一种匹配规则，后面我们会详细介绍。JavaScript 也会在必要的时候自动对 String 原始类型与 String 对象进行转换，这对开发者来说是透明的，你也可以直接使用原始字符串调用字符串对象的属性和方法，示例如下：

```
console.log("Hello".length);//5
```

抽 丝 剥 茧

对于字符串截取方法 slice，如果传入的参数为负数，使用字符串长度减去这个负数的绝对值为最终参数，示例如下：

```
console.log("Hello World".slice(-4,-1));//orl
```

3.2.3　JavaScript 中的 Boolean 对象

Boolean 对象是对布尔类型的原始值的一种包装，同样使用 new 关键字加构造函数的方法来创建，示例如下：

```
var bool = new Boolean(true);
console.log(bool);//[Boolean: true]
```

Boolean 构造函数中所传的参数不一定必须是原始布尔类型的值。注意，如果传入的参数为 0、-0、null、false、NaN、undefined 或者空字符串""，则生成的 Boolean 对象的原始值为 false，其他传入任何值，都将生成一个原始值为 true 的 Boolean 实例对象。示例如下：

```
//以下都将生成原始值为 false 的 Boolean 对象
console.log(new Boolean(0));
console.log(new Boolean(-0));
console.log(new Boolean(NaN));
console.log(new Boolean(undefined));
console.log(new Boolean(""));
console.log(new Boolean(false));
console.log(new Boolean(null));
//下面这些生成的是原始值为 true 的 Boolean 对象
console.log(new Boolean("false"));
console.log(new Boolean({}));
console.log(new Boolean(Infinity));
console.log(new Boolean(new Boolean(false)));
```

在使用 Boolean 实例对象时，有一点需要切记！对于 JavaScript 中的 if 条件语句，其判断条件如果是一个对象则会被自动转换为 true，因此无论 Boolean 实例对象的原始值是 true 还是 false，在 if 条件判断中都将被作为 true 来处理。例如：

```
var bf = new Boolean(false);
if (bf) {
    console.log("执行了");
}
```

上面的示例依然会执行 if 结构中的打印代码。因此，要在判断条件中使用 Boolean 对象，需要取其原始值再进行使用，例如：

```
var bf = new Boolean(false);
if (bf.valueOf()) {
    console.log("执行了");
}
```

如果你只是想将某个其他类型的值转换为布尔类型，直接使用 Boolean()函数即可，转换的规则和前面所讲的一致，0、-0、null、false、NaN、undefined 或者空字符串""将转换为布尔值 false，其他都将转换为布尔值 true。

3.2.4 JavaScript 中的 Array 对象

数组是一种非常常用的数据结构。在 JavaScript 中，数组也是一种对象，只是它是列表形式的对象。简单理解，数组就是一种有序列表，例如我们创建一个学生名单列表，示例如下：

```
var stus = ["Tom","Jaki","Lucy","Ami"];
console.log(typeof stus);//object
```

上面的代码使用中括号进行数组的创建，这是一种便捷创建数组的方法，当然我们也可以使用 Array 构造方法来创建数组对象，这两种方式创建出来的数组对象实质上是一样的，例如：

```
var array = new Array("Tom","Jaki","Lucy","Ami");
console.log(typeof array);//object
```

数组中的元素类型并非要保持一致，它们可以是任意类型的组合。注意，数组中的元素是有序的，可以通过下标的方式对它们进行访问，数组的下标是从 0 开始的，示例如下：

```
//访问数组中的第 1 个元素
console.log(array[0]);//Tom
```

再回过头来看数组对象的构造方法 Array()，这个方法中如果只传入一个参数且为数值类型，则会返回一个固定长度的空数组对象，传入多个参数或非数值类型的参数，则会创建数组并将参数作为数组中的元素。

抽 丝 剥 茧

虽然你也可以将数组理解为类似{0:"Tom", 1:"Jaki"}这样的对象，但是需要注意，访问数组中的元素并不能使用点语法，这样的写法将报错“array.0”。原因是对象的属性名称如果是以数字开头，则其为非法的属性名，不能和点语法结合使用，只能够通过中括号的形式访问。如果创建一个自定义的对象，为其添加一个以数字开头的属性，你依然无法使用点语句进行访问。你可能还会有一个疑问，中括号进行对象属性的访问时，属性名不是必须是字符串格式么？确实如此，只是如果我们传入的是数值，JavaScript 会自动帮我们进行处理。

要判断某一个值是否为数组类型，可以使用 Array 函数对象的 isArray()方法，如果传入的参数是数组对象，则会返回 true，否则将返回 false，示例如下：

```
//判断某个值是否为数组
console.log(Array.isArray(array));//true
```

数组实例对象中也封装了许多属性与方法，其中 length 属性可以获取到数组中元素的个数：

```
var array = new Array("Tom","Jaki","Lucy","Ami");
console.log(array.length);//4
```

需要注意，数组实例对象的 length 属性可以进行任意修改，如果将其修改为小于数组中原本元素个数的值，溢出的元素将会被清掉，因此在修改数组实例对象的 length 属性时要格外注意，示例如下：

```
var array = new Array("Tom","Jaki","Lucy","Ami");
console.log(array.length);//4
array.length = 2;
console.log(array);//[ 'Tom', 'Jaki' ]
```

数组实例中封装的方法可以分为两类，一类是会修改原数组对象的，另一类是不会修改原数组对象的，会将操作结果以新的数组对象返回。下面这些方法都会对原数组对象进行修改：

```
var array = [0,1,2,3,4,5,6,7,8,9];
//删除数组最后一个元素
array.pop();
console.log(array);//[ 0, 1, 2, 3, 4, 5, 6, 7, 8 ]
//在数组的末尾添加元素
array.push(9,10);
console.log(array);//[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
//倒置数组
array.reverse();
console.log(array);//[ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 ]
//删除数组中的第 1 个元素
array.shift();
console.log(array);//[ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 ]
//对数组进行排序
array.sort(function(a,b){
```

```
        if (a>b) {
            return 1;
        }else if(a<b){
            return -1;
        }else{
            return 0;
        }
    });
    console.log(array);//[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
    //进行数组元素替换，第 1 个参数为从哪个下标可以替换，第 2 个参数为要删除的元素个数，
    //之后为要替换成的元素
    array.splice(6,4,"100","end");
    console.log(array);//[ 0, 1, 2, 3, 4, 5, ,'100','end' ]
    //在数组开头进行元素的追加
    array.unshift(0,0);
    console.log(array);//[ 0, 0, 0, 1, 2, 3, 4, 5, '100', 'end' ]
```

在上面列出的方法中，除了 sort()排序方法，其他都很好理解。数组实例对象的 sort()排序方法需要我们传入一个排序函数 function(a,b)，这个函数中有两个参数，分别代表按照数组中的先后顺序进行比较的两个元素，如果排序函数返回小于 0 的值，则表示 a 元素会排在 b 元素之前，如果排序函数返回大于 0 的值，则表示 a 元素要排在 b 元素之后，如果排序函数返回 0，则 a 元素和 b 元素的相对位置不变。

下面这些方法不会修改原数组实例对象：

```
    var array = [0,1,2,3,4,5];
    //进行数组元素追加
    console.log(array.concat(6,7,8,9));//[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
    //将所有元素以传入的参数分隔进行拼接
    console.log(array.join("."));//0.1.2.3.4.5
    //截取子数组，第 1 个参数为开始截取的下标，第 2 个参数为截取到的下标(不包含此位置)
    console.log(array.slice(0,3));//[ 0, 1, 2 ]
    //将数组元素拼接成字符串，以逗号分隔
    console.log(array.toString());//0,1,2,3,4,5
    //返回数组中指定元素的下标，从前往后找
    console.log(array.indexOf(1));//1
    //返回数组中指定元素的下标，从后往前找
    console.log(array.lastIndexOf(1));//1
```

需要注意，数组中的元素是可以重复的。

在实际开发中，对数组最频繁的操作便是进行数组的遍历。JavaScript 中的数组实例对象支持很多遍历方法，例如：

```
    var stus = ["Tom","Jaki","Lucy","Ami"];
    //进行数组的逐个遍历，需要传入一个函数，
    //此函数有 3 个参数，分别为遍历到的元素、其下标和原数组。
    //还可以传入第 2 个参数，这个参数将与遍历函数中使用的 this 关联
```

```
stus.forEach(function(element,index,array){
    console.log(element,index,array);
},stus);
//对数组进行检查。传入一个函数，函数中的参数分别为遍历到的元素、
//其下标和原数组。此函数需要返回一个布尔值，如果返回 ture，则继续遍历，
//如果返回 false 则停止。
//当所有元素都遍历完成并且都返回 true 时，结果才为 true，否则都为 false
//every 方法还可以传入第 2 个参数，这个参数将与遍历函数中使用的 this 关联
var notHaveAmi = stus.every(function(element,index,array){
    console.log(this);
    console.log(element,index,array);
    if (element == "Ami") {
        return false;
    }else{
        return true;
    }
},stus);
console.log(notHaveAmi);//false
//some 遍历方法与 every 对应，
//只是其回调是全部返回 false，结果才为 false，否则为 true
var haveAmi = stus.some(function(element,index,array){
    console.log(this);
    console.log(element,index,array);
    if (element == "Ami") {
        return true;
    }else{
        return false;
    }
},stus);
console.log(haveAmi);//true
//数组过滤器。当回调函数返回 true 时，代表此元素通过过滤，将其添加进新数组返回
var newArray = stus.filter(function(element,index,array){
    console.log(this);
    console.log(element,index,array);
    if (element == "Ami") {
        return true;
    }else{
        return false;
    }
},stus);
console.log(newArray);//[ 'Ami' ]
//map 方法将数组中的每一个元素执行回调，然后将返回值重新组成数组返回
newArray = stus.map(function(element,index,array){
    return element+"!";
},stus);
```

```
console.log(newArray);//[ 'Tom!', 'Jaki!', 'Lucy!', 'Ami!' ]
//reduce 方法是数组累加器，会按照数组从左向右的顺序依次调用回调函数。
//回调函数中有 3 个参数，第 1 个参数为上次执行累加回调的返回值，
//第 2 个参数为当前遍历的元素，第 3 个参数为其下标。
//reduce 方法的第 2 个参数可选，为首次进行累加回调的初始值。
//如果不提供初始值，
//reduce 函数会从数组索引为 1 的位置开始，跳过第 1 个元素
var res = stus.reduce(function(acc,element,index){
    return acc+" "+element;
},"Hello");
console.log(res);//Hello Tom Jaki Lucy Ami
//与 reduce 方法类似，其从右向左开始遍历
res  = stus.reduceRight(function(acc,element,index){
    return acc+" "+element;
},"Hello");
console.log(res);//Hello Ami Lucy Jaki Tom
```

抽 丝 剥 茧

虽然 JavaScript 中并没有严格要求在数组的遍历过程中不能对数组结构进行修改，但是建议在遍历的过程中不要增删元素，如果在遍历过程中向数组中新增元素，则新的元素不会被遍历到，如果进行了元素的删除，则会造成意料之外的结果。

上面列举的数组实例对象遍历方法略微复杂，虽然有详尽的注释但是依然强烈建议你一定要自己动手练习一下，实践出真知，熟练掌握这些遍历方法，会让你以后的学习得心应手！

3.2.5 JavaScript 中的 Date 对象

Date 对象是 JavaScript 中用来处理日期与时间的，同样，使用 Date 构造方法来进行 Date 实例对象的创建，在创建 Date 实例对象时有 4 种传参方式，示例如下：

```
//以当前时间创建 Date 对象
var date = new Date();
console.log(date);//2017-02-27T02:11:03.945Z
//以指定时间戳创建 Date 对象，时间戳单位为毫秒
date = new Date(1483888888999);
console.log(date);//2017-01-08T15:21:28.999Z
//以指定的字符串创建 Date 对象，字符串必须为标准格式的时间字符串
date = new Date("December 17, 2017 03:24:00");
console.log(date);//2017-12-16T19:24:00.000Z
//设置年、月、日、时、分、秒、毫秒来创建 Date 对象
date = new Date(2017,10,11,8,10,40,133);
console.log(date);//2017-11-11T00:10:40.133Z
```

如果在构造函数中不传入任何参数，则默认使用当前日期时间创建 Date 实例对象。如果传入一个数值类型的参数，则会将此数值作为时间戳来创建 Date 实例对象，时间戳的单位为毫秒，代

表从 1970 年 1 月 1 日起经过的毫秒数。如果传入的参数是一个字符串，则会以此字符串表示的日期时间创建 Date 对象。需要注意，传入的参数字符串必须为标准的日期时间字符串。如果传入的参数大于 1 个，则会按照年、月、日、时、分、秒、毫秒的顺序进行读参来创建 Date 对象。如果传入的参数在 1 个以上但是不足 7 个，则未传入的参数将以 0 代替，代表月数的整数值需是 0～11 的一个数值，0 代表 1 月。

使用 Date 构造函数对象的 now 方法可以直接获取到 1970 年 1 月 1 日至当前时刻的时间戳，单位为毫秒，示例如下：

```
//返回从 1970 年 1 月 1 日起至现在经过的毫秒数
console.log(Date.now());//1488162786627
```

Date 构造函数对象的 parse 方法用于解析一个日期时间字符串，将返回 1970 年 1 月 1 日至指定日期时间的时间戳，单位为毫秒，示例如下：

```
//解析一个日期时间字符串
console.log(Date.parse("December 17, 2017 03:24:00"));//1513452240000
```

Date 构造函数对象的 UTC 方法用于将指定的日期时间转换为时间戳，其参数规则与 Date()构造函数中最长参数的形式一致，最多可以接收 7 个参数，示例如下：

```
//指定日期时间，返回时间戳
console.log(Date.UTC(2017,0,1,10,30,30,120));//1483266630120
```

你也可以使用 Date 实例对象中封装的一些方法来直接获取想要的信息，示例如下：

```
var date = new Date();
//根据本地时间获取日期对象是当前月的第几天
console.log(date.getDate());
//根据本地时间获取星期。从 0 开始，0 表示周日
console.log(date.getDay());
//根据本地时间获取日期对象当前的年份
console.log(date.getFullYear());
//根据本地时间获取日期对象当前的小时，0～23
console.log(date.getHours());
//根据本地时间获取日期对象当前的分钟
console.log(date.getMinutes());
//根据本地时间获取日期对象当前的秒数
console.log(date.getSeconds());
//根据本地时间获取日期对象当前的毫秒数
console.log(date.getMilliseconds());
//根据本地时间获取日期对象当前的月份。从 0 开始，0 表示 1 月
console.log(date.getMonth());
//返回时间戳，单位毫秒
console.log(date.getTime());
//获取当前时区的时区偏移
console.log(date.getTimezoneOffset());
//根据通用时间获取当前日期对象是当前月的第几天
```

```
console.log(date.getUTCDate());
//根据通用时间获取当前日期对象的星期数，0 表示周日
console.log(date.getUTCDay());
//根据通用时间获取日期对象当前的年份
console.log(date.getUTCFullYear());
//根据通用时间获取日期对象当前的小时
console.log(date.getUTCHours());
//根据通用时间获取日期对象当前的分钟
console.log(date.getUTCMinutes());
//根据通用时间获取日期对象当前的秒数
console.log(date.getUTCSeconds());
//根据通用时间获取日期对象当前的毫秒数
console.log(date.getUTCMilliseconds());
//根据通用时间获取日期对象当前的月份，从 0 开始，0 表示 1 月
console.log(date.getUTCMonth());
```

博 闻 强 识

UTC 是协调世界时的简称，又称为世界统一时间或世界标准时间，被应用于许多互联网标准中。

下面这些方法用来修改 Date 实例对象：

```
var date = new Date();
//根据本地时间为日期对象设置月份中的第几天
date.setDate(10);
//根据本地时间为日期对象设置年份
date.setFullYear(1999);
//根据本地时间为日期对象设置小时
date.setHours(11);
//根据本地时间为日期对象设置毫秒数
date.setMilliseconds(123);
//根据本地时间为日期对象设置分钟数
date.setMinutes(30);
//根据本地时间为日期对象设置月份
date.setMonth(1);
//根据本地时间为日期对象设置秒数
date.setSeconds(30);
//根据时间戳来设置日期对象的时间，如果早于 1970 年 1 月 1 日，可以设置为负值
date.setTime(1488167242644);
//根据通用时间为日期对象设置月份中的第几天
date.setUTCDate(10);
//根据通用时间为日期对象设置年份
date.setUTCFullYear(1970);
//根据通用时间为日期对象设置毫秒数
date.setUTCMilliseconds(123);
```

```
//根据通用时间为日期对象设置分钟数
date.setUTCMinutes(30);
//根据通用时间为日期对象设置月份
date.setUTCMonth(1);
//根据通用时间为日期对象设置秒数
date.setUTCSeconds(30);
```

下面的这些方法用于对 Date 实例对象进行格式化：

```
var date = new Date();
//以一种易读的方式返回日期
console.log(date.toDateString());//Mon Feb 27 2017 星期 月 日 年
//返回符合 ISO 标准的日期字符串
console.log(date.toISOString());//2017-02-27T05:13:25.025Z
//使用 toISOString() 返回一个表示该日期的字符串
console.log(date.toJSON());//2017-02-27T05:14:11.414Z
//返回一个表示该日期对象日期部分的字符串，该字符串格式与系统设置的地区关联
console.log(date.toLocaleDateString());//2/27/2017
//返回一个表示该日期对象的字符串，该字符串与系统设置的地区关联
console.log(date.toLocaleString());//2/27/2017, 1:16:27 PM
//返回一个表示该日期对象时间部分的字符串，该字符串格式与系统设置的地区关联
console.log(date.toLocaleTimeString());//1:17:00 PM
//返回一个表示该日期对象的字符串
console.log(date.toString());//Mon Feb 27 2017 13:17:48 GMT+0800 (CST)
//以人类易读格式返回日期对象时间部分的字符串
console.log(date.toTimeString());//13:18:19 GMT+0800 (CST)
//把一个日期对象转换为一个以 UTC 时区计时的字符串
console.log(date.toUTCString());//Mon, 27 Feb 2017 05:18:43 GMT
//从 1970 年 1 月 1 日 0 时 0 分 0 秒（UTC，协调世界时）到该日期的毫秒数，与 getTime()方法一致
console.log(date.valueOf());
```

注意，如果在设置 Date 实例对象的年份时使用了两位数，则将默认表示为 1900－1999 年之间的年份，例如：

```
var date = new Date(90,9,22);
console.log(date);//1990-10-21T16:00:00.000Z
```

为了避免不必要的歧义，最好不要使用两位数来设置 Date 实例对象的年份。

3.2.6　JavaScript 中的 Math 对象

在编程过程中，数学运算是其中十分重要的角色，毕竟计算机学科的基础是基于数学的。在各种编程语言中也都提供了数学函数库供开发者使用，在 JavaScript 中，Math 是一个内置对象而并非函数对象。这和我们之前学习的内置对象有些不同，也很容易理解，Math 对象的作用是提供便利的数学方法，我们并不需要创建出实例对象。

在开发中可能经常会用到一些数学常量，例如圆周率、自然对数等。Math 对象中包装了一些属性可以直接获取到这些常量，示例如下：

```
//数学常量
//自然常数
console.log(Math.E);//2.718281828459045
//2 的自然对数
console.log(Math.LN2);//0.6931471805599453
//10 的自然对数
console.log(Math.LN10);//2.302585092994046
//以 2 为底 E 的对数
console.log(Math.LOG2E);//1.4426950408889634
//以 10 为底 E 的对数
console.log(Math.LOG10E);//0.4342944819032518
//圆周率
console.log(Math.PI);//3.141592653589793
//1/2 的平方根
console.log(Math.SQRT1_2);//0.7071067811865476
//2 的平方根
console.log(Math.SQRT2);//1.4142135623730951
```

也可以使用 Math 对象中提供的方法来进行数学运算，常用方法列举如下：

```
//数学方法
//求绝对值
console.log(Math.abs(-4));//4
//求反余弦值
console.log(Math.acos(0.5));//1.0471975511965976
//求反正弦值
console.log(Math.asin(0.5));//0.5235987755982988
//求反正切值
console.log(Math.atan(0.5));//0.46364760900080615
//需要传入两个参数 x、y，求 x/y 的反正切值
console.log(Math.atan2(1,2));//0.4636476090008061
//进行向上取整
console.log(Math.ceil(1.1));//2
//求余弦值
console.log(Math.cos(0.5));//0.8775825618903728
//传入参数 n 求自然对数 E 的 n 次方
console.log(Math.exp(2));//7.38905609893065
//向下取整
console.log(Math.floor(2.9));//2
//传入参数 n，求以自然常数 E 为底 n 的对数
console.log(Math.log(10));//2.302585092994046
//求一组数中的最大值
console.log(Math.max(1,2,5,3,7));//7
```

```
//求一组数中的最小值
console.log(Math.min(1,2,5,3,7));//1
//传入两个参数 x、y，求 x 的 y 次方
console.log(Math.pow(2,3));//8
//返回一个 0～1 的随机数
console.log(Math.random());
//进行四舍五入
console.log(Math.round(3.4));//3
//求正弦值
console.log(Math.sin(0.5));
//求平方根
console.log(Math.sqrt(2));//1.4142135623730951
//求正切值
console.log(Math.tan(1));
```

需要注意，上面所列举的三角函数方法的返回值都是以弧度为单位的。

3.2.7　JavaScript 中的 RegExp 对象

关于正则表达式，我们前边在介绍 String 对象时已经提到，正则表达式用来定义一种规则，通过规则来对文本进行匹配。在 JavaScript 中，正则表达式也是一种对象，其可以使用 RegExp 构造函数来创建。

你可以通过两种方式创建正则表达式对象，最简单的方式是通过字面量来创建正则表达式，示例如下：

```
//通过字面量来创建正则表达式
var reg = /hello/i;
```

通过字面量创建正则表达式有这样的规则：斜杠符内编写正则表达式文本，结束斜杠的右侧指定匹配模式。匹配模式可以是表 3-1 中几种标志的组合。

表 3-1　匹配模式标志

匹配模式标志	作用
g	全局匹配（匹配到一个结果后还会继续向后匹配直到结束）
i	匹配过程忽略大小写
m	多行匹配模式（默认情况下，开始符^与结束符$是工作在单行模式的，将只匹配整个文本的开始与结束，配置这个参数后，其会匹配每一行的开始与结束。）

例如，我们对“Hello world hello”进行不区分大小写的全局匹配，将会匹配到“Hello”和“hello”，示例如下：

```
var reg = /hello/ig;
var res = "Hello world hello".match(reg)
console.log(res);//[ 'Hello', 'hello' ]
```

使用 RegExp 构造函数对象来创建正则表达式对象，示例如下：

```
var reg2 = new RegExp("hello",'ig');
console.log("Hello world hello".match(reg2));//[ 'Hello', 'hello' ]
```

RegExp()构造函数中可以传入两个参数（第 2 个参数可选），第 1 个参数为正则表达式字符串，第 2 个参数为匹配模式。

抽 丝 剥 茧

RegExp 构造方法中的第 1 个参数可以传入字符串形式的正则表达式，也可以直接传入字面量语法的正则表达式，例如如下的语法也是正确的：

```
var reg2 = new RegExp(/hello/,'ig');
console.log("Hello world hello".match(reg2));//[ 'Hello', 'hello' ]
```

需要额外注意，当 RegExp 构造方法中第 1 个参数为字符串时，如果此字符串中有特殊字符则需要进行转义，例如如下两个正则对象是完全等价的：

```
var re = new RegExp("\\w+");
var re = /\w+/;
```

关于正则表达式，还有很多东西我们可以深入探究一下。正则表达式又称规则表达式，通过一些符号的组合来定义一种搜索算法。简单理解，一个正则表达式就是一个逻辑公式，通过公式来对文本进行精确或模糊搜索。正则表达式中定义了一些特殊字符，大致可以分为 4 类：字符类别、字符集合、边界、数量词。

字符类别用于模糊匹配，即某一个特殊字符可以代表某一类字符。常用的特殊字符列表如表 3-2 所示。

表 3-2　常用的特殊字符列表

字符	含义	示例
.（小数点）	匹配任意字符（换行符除外）	将匹配 hell： reg = /h..l/; "hello".match(reg)
\d	匹配 0～9 的任何一个数字字符	将匹配 5h： reg = /\dh/; "5hello".match(reg);
\D	匹配任意一个不是数字的字符	将匹配 Eh： reg = /\Dh/; "Ehello".match(reg)
\w	匹配一个字母、数字或下画线字符	将匹配 he： reg = /h\w/; "Ehello".match(reg);
\W	匹配任意非字母、非数字和非下画线的字符	将匹配 h$： reg = /h\W/; "h$llo".match(reg);

（续表）

字符	含义	示例
\s	匹配一个空白符，包括空格、制表符、换页符、换行符等	将匹配 h e： reg = /h\se/; "h ello".match(reg);
\S	匹配任意一个非空白字符	将匹配 he： reg = /h\S/; "hello".match(reg);
\t	匹配一个水平制表符	
\r	匹配一个回车符	
\n	匹配一个换行符	
\v	匹配一个垂直制表符	
\f	匹配一个换页符	
[\b]	匹配一个退格符	
\0	匹配一个 NUL 字符	
\xhh	匹配编码为 hh 的字符（十六进制）	
\uhhhh	匹配 Unicode 值为 hhhh 的字符	

字符集合用于匹配某个集合内的字符，如表 3-3 所示。

表3-3　字符集合

字符集合	含义	示例
[abc]	匹配中括号内的一个字符	将匹配 he： reg = /h[abced]/; "hello".match(reg);
[a-b]	匹配范围内的字符	将匹配 he： reg = /h[a-e]/; "hello".match(reg);
[^abc]	匹配除了集合字符外的所有字符	
[^a-b]	匹配除集合范围外的所有字符	

用来定义边界的特殊字符如表 3-4 所示。

表3-4　定义边界的特殊字符

字符	含义	示例
^	匹配字符串的开头	将匹配到 h： reg = /^h/; "hello".match(reg);
$	匹配字符串的结尾	将匹配到 o： reg = /o$/; "hello".match(reg);

用来设置匹配字符数量的特殊字符如表 3-5 所示。

表3-5　设置匹配字符数量的特殊字符

字符	含义	示例
x*	匹配 0 次或多次 x 字符	将匹配到 ell： reg = /el*/; "hello".match(reg);
x+	匹配 1 次或多次 x 字符	将匹配到 ell： reg = /el+/; "hello".match(reg);
x?	匹配 0 次或 1 次 x 字符	
x(?!y)	当 x 字符后面不是 y 字符时才匹配	将匹配到 lo： reg = /l(?!l)./; "hello".match(reg);
x|y	匹配字符 x 或者字符 y	
x{n}	匹配连续 n 个 x	将匹配到 ll： reg = /l{2}/; "hello".match(reg);
x{n,}	匹配至少连续 n 个 x	
x{n,m}	匹配连续出现 n～m 次个 x	

上面我们使用了大量的篇幅来介绍有关正则表达式的基础知识，这是十分有必要的。正则表达式广泛应用于用户表单的验证中，例如验证用户输入的邮箱是否合法、用户输入的手机号是否合法等。再回到 RegExp 实例对象，其中的一些属性可以用来获取设置的正则匹配模式，示例如下：

```
var reg = new RegExp(/l./);
//是否开启全局匹配模式
console.log(reg.global);//false
//是否开启忽略大小写
console.log(reg.ignoreCase);//false
//是否开启多行模式
console.log(reg.multiline);//false
```

RegExp 实例对象中定义如下的方法，可以直接对一个目标字符串进行检测：

```
var reg = new RegExp(/l./);
//对目标字符串进行正则匹配
console.log(reg.exec("hello"));//[ 'll', index: 2, input: 'hello' ]
//检测目标字符串是否可以通过正则验证，即匹配到结果
console.log(reg.test("hello"));//true
```

在实际开发中 RegExp 实例对象的 test 方法要比 exec 方法更加常用，很多情况下，我们都需要用它来验证一个字符串是否符合规则。

3.2.8　JavaScript 中的 Function 对象

前面的章节中已经对函数有过简单介绍。我们知道，函数实际上也是一种对象，其是由 Function 构造函数创建出来的。Function 实例对象中的 arguments 属性可以获取到调用函数时所有传递进函数的实参，length 属性可以获取当前函数的形参个数。本小节将介绍几个 Function 实例对象的方法，这些方法在 JavaScript 面向对象技术中十分重要。

在 JavaScript 中，this 是一个十分重要的关键字，也是略微难于理解的关键字。在函数内部，this 的值取决于函数是如何调用的，直接调用全局函数，则函数内部的 this 指向全局对象。如果函数作为某个对象的行为进行调用，则其中的 this 指向调用函数的对象，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(){
        console.log("姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString();//姓名：Jaki、年龄：25
```

toString 方法是由 teacher 对象调用的，toString 方法中的 this 就代表当前 teacher 对象本身。在构造函数中使用 this 则情况不同，new 关键字会创建一个空的对象，然后将构造函数中的 this 和这个对象进行绑定。

了解了 this 关键字，我们再来看 Function 实例对象中的几个方法就十分容易理解了。Function 实例对象中的 apply 方法用于指定调用方法的 this 值和参数。我们把前边的代码略作修改，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
        console.log(owner+"姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString("Teacher");//Teacher 姓名：Jaki、年龄：25
var student = {
    name:"Lucy",
    age:23
}
teacher.toString.apply(student,["Student"]);//Student 姓名：Lucy、年龄：23
```

上面的示例代码中创建了两个对象 teacher 与 student。student 对象并没有 toString 方法，但是我们可以借助 teacher 对象的 toString 方法来打印 student 对象的信息。Function 实例对象中提供了 apply 方法，这个方法可以接收两个参数，第 1 个参数设置调用方法的上下文，即函数中 this 关键

字的指向，第 2 个参数为一个数组，数组中的元素将被作为实参传入函数。有了 apply 方法，我们可以实现某个对象调用其他对象的方法。

与 apply 方法十分类似的还有 call 方法，其作用也是设置所调用函数中 this 的指向与传入参数，不同的是 call 函数中一参数个数不一定，第 1 个参数为要绑定到 this 的对象，之后的所有参数都会作为实参传入函数，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
        console.log(owner+"姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString("Teacher");//Teacher 姓名：Jaki、年龄：25
var student = {
    name:"Lucy",
    age:23
}
teacher.toString.call(student,"Student");//Student 姓名：Lucy、年龄：23
```

Function 实例对象中还提供了一个 bind 方法，这个方法将会创建一个新的函数，bind 方法中第 1 个参数为新创建函数调用时的 this 指向，之后所有的参数都将作为默认内置实参传入函数，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
        console.log(owner+"姓名："+this.name+"、年龄："+this.age);
    }
}
teacher.toString("Teacher");//Teacher 姓名：Jaki、年龄：25
var student = {
    name:"Lucy",
    age:23
}
var studentToString = teacher.toString.bind(student,"Student");
studentToString();//Student 姓名：Lucy、年龄：23
```

需要注意，bind 函数中定义的参数在传入新生成的函数时会被置为内置参数放在所有实参之前，不会被覆盖掉，示例如下：

```
var teacher = {
    name:"Jaki",
    age:"25",
    toString:function(owner){
```

```
            console.log(owner+"姓名："+this.name+"、年龄："+this.age);
            console.log(arguments);
        }
    }
    teacher.toString("Teacher");//Teacher 姓名：Jaki、年龄：25
    var student = {
        name:"Lucy",
        age:23
    }
    var studentToString = teacher.toString.bind(student,"Student");
    //Student 姓名：Lucy、年龄：23
    //{ '0': 'Student', '1': 'Hello' }
    studentToString("Hello");
```

3.3　深入 JavaScript 中的 Object 对象

在 JavaScript 中，Object 构造函数可以理解为一个对象包装器，使用这个构造函数可以创建出最原始的实例对象。我们知道，在 JavaScript 中自定义对象有两种方式，示例如下：

```
    //使用字面量语法创建对象
    var teacher = {
        name:"Jaki",
        age:25,
        teaching:function(){
            console.log("teching...");
        }
    };
    //使用 Object 构造函数创建对象
    var student = new Object();
    student.name = "Lucy";
    student.age = 24;
    student.learning = function(){
        console.log("learning...");
    };
```

当我们使用 Object()构造函数来进行对象的创建时，里面可以传入一个参数，构造函数会根据传入的参数类型创建相应的对象实例。

如上我们定义的对象都可以直接通过点语法或者中括号的方式进行属性和方法的访问，同样我们也可以自由地对对象中的属性方法进行枚举、修改、删除等。实际上，对象中的属性和方法都有一套配置参数，这些配置参数决定了对象的可读性、可枚举性和可配置性等。

3.3.1 为对象属性进行配置

Object 构造方法对象中提供了一个 defineProperty()方法，这个方法会直接在一个对象上定义一个新的属性或者修改一个已经存在的属性。示例如下：

```
//使用 Object 构造函数创建对象
var student = new Object();
student.name = "Lucy";
student.age = 24;
student.learning = function(){
    console.log("learning...");
};
Object.defineProperty(student,"name",{
    configurable:true,
    enumerable:true,
    writable:true,
    value:"July"
});
console.log(student.name);//July
```

上面的代码使用 defineProperty()方法对 student 对象的 name 属性进行了重新配置，这个方法中需要传入 3 个参数，第 1 个参数为要进行配置的对象，第 2 个参数为需要进行配置的属性名，第 3 个参数为配置描述参数。

下面解释配置描述中可选的 4 个配置项所代表的意义。configurable 用来设置此属性是否是可配置的，当第一次使用 defineProperty 方法对某个属性进行配置时，如果将此配置项设置为 false，则之后不可以再对对象的这个属性配置进行修改。enumerable 配置此属性的可枚举性，如果设置为 true，则此属性可以被 for-in 结构遍历到；如果配置为 false，则此属性不能被 for-in 结构遍历到。writeable 配置此属性的可写性，如果设置为 true，则此属性可以被赋值运算符修改；如果配置为 false，则此属性不能被赋值运算符修改。value 项其实就是设置当前属性的值。

也可以在 defineProperty()方法的描述参数中定义 getter 与 setter 方法。getter 方法的返回值会被作为该属性的值，setter 方法则是当属性被赋值时被调用，示例如下：

```
//使用 Object 构造函数创建对象
var student = new Object();
student.name = "Lucy";
student.age = 24;
student.learning = function(){
    console.log("learning...");
};
var name = "Lucy";
Object.defineProperty(student,"name",{
    configurable:true,
    enumerable:true,
```

```
        get:function(){
            console.log("正在使用 name 属性");
            return name;
        },
        set:function(value){
            console.log("将要设置 name 属性");
            name = value;
        }
    });
    console.log(student.name);//正在使用 name 属性 Lucy
    student.name = "July";//将要设置 name 属性
    console.log(student.name);//正在使用 name 属性 July
```

需要注意，描述参数中的 value、writable 两个配置项与 get、set 两个配置项不能同时存在，否则 JavaScript 代码在运行时会抛出异常。有了属性的 get 和 set 配置，我们可以方便地监听对象某个属性的设置或获取，添加需要的业务逻辑。

Object 构造函数对象中还提供了一个方法，可以一次定义或修改多个属性，示例如下：

```
    var teacher = {
        name:"Jaki",
        age:25,
        teaching:function(){
            console.log("teching...");
        }
    };
    Object.defineProperties(teacher,{
        "name":{
            value:"珲少",
            writable:false
        },
        "age":{
            value:25,
            writable:false
        }
    });
```

3.3.2　Object 构造方法对象中的常用函数

除了前面提到的配置属性方法外，Object 构造函数对象中还有一些其他针对实例对象操作的方法，其中 assign 方法可以实现将几个目标对象中可枚举的属性复制到源对象中，示例如下：

```
    var teacher = {
        name:"jaki",
        age:24,
        teaching:function(){
```

```
            console.log("teaching");
        }
    };
    var teacher2 = {
        subject:"JavaScript"
    }
    Object.defineProperty(teacher,"number",{
        value:1001,
        enumerable:false
    });
    console.log(teacher.number);//1001
    for(prop in teacher){
        console.log(prop);
    }
    //进行对象可枚举属性的复制
    var obj = {};
    //第 1 个参数为目标对象，其后的参数为要被复制属性的对象
    Object.assign(obj,teacher,teacher2);
    console.log(obj.name+obj.age+obj.subject);//jaki24JavaScript
    obj.teaching();//teaching
    console.log(obj.number);//undefined
```

上面的代码将 teacher 对象和 teacher2 对象中的可枚举属性全部复制进了 obj 对象中，number 属性是不可枚举的，因此此属性并没有进行复制，除此之外，从原型继承来的属性也不会被复制（原型与继承在后面有具体介绍）。需要注意，在进行属性复制时，源对象不可以是 null，如果源对象中的属性名和目标对象的属性名重复，则源对象的属性值会被覆盖。还有一点需要注意，assign 方法进行的复制都是浅复制，即如果目标对象的某个属性值是引用类型，则它复制的是此引用，并不是引用所对应的值，示例如下：

```
    //深浅复制
    var obj1 = {
        a:{
            name:"Jaki"
        },
        b:25
    };
    var obj2 = {};
    Object.assign(obj2,obj1);
    //修改 obj1
    obj1.a.name = "Lucy";
    obj1.b = 23;
    //obj2 中的 b 并没有被修改，因为其是原始值类型，但是 a 属性被修改了，因为其是引用类型
    console.log(obj2);//{ a: { name: 'Lucy' }, b: 25 }
```

抽 丝 剥 茧

深浅复制是一个比较难于理解的问题，如果有疑惑，你可以回顾本书第 1 章内容中关于原始类型与引用类型的相关内容。

Object 构造方法中的 create 函数也是用于创建一个对象，其创建对象的时候可以指定对象的原型和若干属性。这里你不需要对对象的原型做过多深入研究，简单理解，原型是对象所继承的对象，对象可以直接使用原型中定义的属性。例如，对于所有讲 JavaScript 教程的教师，我们可以定义一个父对象，其中定义一个所教课程的属性，其他由此对象继承而来的子对象中都可以使用这个属性，示例代码如下：

```
//继承的实现
var base = {
    subject:"JavaScript"
}
var teacher1 = Object.create(base,{
    "name":{
        value:"Jaki",
        enumerable:true
    },
    "age":{
        value:25,
        enumerable:true
    }
});
console.log(teacher1);//{ name: 'Jaki', age: 25 }
console.log(teacher1.subject);//JavaScript
```

需要注意，create 方法中的两个参数，第 1 个参数为所创建对象的原型，第 2 个参数为要创建对象的属性配置列表，和 defineProperties 方法中第 2 个参数的含义一致。

Object 构造函数中的 freeze 函数用于冻结对象。冻结一词指的是不能向对象中添加新的属性，不能修改或者删除对象的属性，同样也不能对对象中属性的配置进行修改。简单理解，冻结的对象是一个完全不可变的对象。freeze 方法传入一个对象作为参数，然后返回被冻结的对象。示例如下：

```
var fre = {
    name:"Jaki"
};
fre = Object.freeze(fre);
fre.name = "Lucy";
//冻结的对象不能修改
console.log(fre);//{ name: 'Jaki' }
```

Object 构造方法中的 isFrozen 方法可以获取某个对象是否为被冻结的对象。

Object 构造函数对象中的 getOwnPropertyDescriptor 函数用来获取对象某个属性的配置信息，示例如下：

```
var fre = {
    name:"Jaki"
};
/*
{ value: 'Jaki',
  writable: true,
  enumerable: true,
  configurable: true }
*/
var des = Object.getOwnPropertyDescriptor(fre,"name");
```

Object 构造函数对象中的 getOwnPropertyNames 函数可以获取指定对象所有自身的属性，从原型继承来的属性不包括在内。需要注意，这个方法会将可枚举和不可枚举的属性都获取到。示例如下：

```
var base = {
    subject:"JavaScript"
}
var teacher1 = Object.create(base,{
    "name":{
        value:"Jaki",
        enumerable:true
    },
    "age":{
        value:25,
        enumerable:true
    }
});
console.log(Object.getOwnPropertyNames(teacher1));//[ 'name', 'age' ]
```

Object 构造函数对象中 getPrototypeOf 方法可以获取到某个对象的原型对象，示例如下：

```
var base = {
    subject:"JavaScript"
}
var teacher1 = Object.create(base,{
    "name":{
        value:"Jaki",
        enumerable:true
    },
    "age":{
        value:25,
        enumerable:true
    }
});
console.log(Object.getPrototypeOf(teacher1));//{ subject: 'JavaScript' }
```

Object 构造函数对象中的 seal 方法用来密封对象。密封对象是指不能添加新属性，不能删除已有属性，不能修改属性的配置，但是可以修改属性的值的对象。密封与冻结的唯一不同是对象属性的值是否可以修改。示例如下：

```
var seal = {
    name:"Jaki"
};
self = Object.seal(seal);
//密封对象不能添加新属性
seal.age = 25;//undefined
console.log(seal.age);
```

isSealed 方法用来判断一个对象是否是密封的。

除了冻结与密封，对象还有“扩展”的概念。可扩展表示对象可以添加新的属性，不可扩展表示对象不能够添加新的属性，但是可以删除和修改已有的属性，Object 构造方法中也有函数对对象的可扩展性进行控制，perventExtensions 方法用于约束对象的扩展性，示例如下：

```
var ext = {
    name:"Jaki"
}
//抑制对象扩展
ext = Object.preventExtensions(ext);
ext.age = 25;
console.log(ext.age);//undefined
```

同样也有 isExtensible 方法用来判断某个对象是否是可扩展的。

Object 构造方法对象中还定义了一个 keys 方法，其会返回对象自身可枚举的属性名。需要注意，其和 for-in 遍历结构还是有一些区别的，keys 方法只会返回对象自身的可枚举属性，for-in 遍历还可以遍历出对象原型链上的可枚举属性。示例如下：

```
var base = {
    subject:"JavaScript"
}
var teacher1 = Object.create(base,{
    "name":{
        value:"Jaki",
        enumerable:true
    },
    "age":{
        value:25,
        enumerable:true
    }
});
console.log(Object.keys(teacher1));//[ 'name', 'age' ]
```

抽 丝 剥 茧

直接打印对象实际上会打印出对象中可枚举的属性。

3.3.3 Object 实例对象中的常用方法

Object 实例对象中也有一些方法，主要与其属性的检测相关。hasOwnProperty 方法可以用来检查对象中是否包含某个属性，此属性必须是对象本身的，不能是从原型链上继承来的，示例如下：

```
//判断某个对象本身是否包含指定的属性，此属性不是原型链上的
console.log(teacher.hasOwnProperty("name"));//true
```

isPrototypeOf 方法用来检查当前对象是否在某个对象的原型链上，示例如下：

```
var teacher = new Object();
var prototype =  {
    subject:"JavaScript"
};
//设置原型
Object.setPrototypeOf(teacher,prototype);
teacher.name = "Jaki";
teacher.age = 25;
teacher.teaching = function(){
    console.log("teaching");
}
console.log(prototype.isPrototypeOf(teacher));//true
```

propertyIsEnumerable 方法用来检查对象的某个属性是否是可枚举的，也将返回布尔值，示例如下：

```
var prototype =  {
    subject:"JavaScript"
};
//检查对象的某个属性是否为可枚举的
console.log(prototype.propertyIsEnumerable("subject"));//true
```

3.4 面向对象编程技术

面向对象编程是一种计算机编程框架，简称 OOP（Object Oriented Programming）。面向对象编程的基本原则是程序是由独立作用的对象组成的，使用面向对象编程技术开发的软件有着更好的重用性、灵活性和扩展性。

学习面向对象编程技术，首先需要了解下面几个概念。

1. 对象

对象是面向对象编程的核心，没有对象就没有办法面向对象。对象是可以实现某些功能的小单元，对象中会封装属性与行为。在 JavaScript 语言中，行为也可以理解为是一种属性。属性的实质就是变量或常量，只是其有特殊意义并与此对象相关，行为的实质就是方法，即函数，用来实现对象的某些功能。

2. 类

面向对象实际上也是自然生活的一种模拟，生活中存在各种各样的事物，一棵树是一个对象，一辆汽车是一个对象，每个人也是一个对象，等等。我们会将对象进行分门别类，例如生物下面分为动物和植物，动物下面又分为鱼类、鸟类、人类等，对于人类，我们又可以分为男人和女人，男人里面我们可以再细分，如老年、中年、青年和少年。将世间万物分类的基础在于同一类事物有统一的属性和类似的行为，例如鸟类都有翅膀，可以飞翔。这样的分类可以使我们更轻松地了解自然，管理事物。在程序的世界里也是如此，前面我们一直在拿教师对象做示例，教师就可以定义为一个类，所有的教师都有姓名、年龄、所教科目这些属性，只是这些属性的值可能不同。

3. 封装

封装是将属性和行为捆绑在一起，创建对象的过程。

4. 继承

继承描述的是一种关系，表示类的从属关系。在继承体系中子类会继承父类的属性和行为，子类也可以重新定义自己的属性和行为，同样，子类也可以修改从父类继承来的属性和行为。这很像现实生活中的父子关系，孩子会继承父亲的一些外貌、性格等，但是孩子也有许多自己形成的个性。

5. 组合

组合简单理解就是一个类可以作为另一个类的属性。例如，一辆汽车会由很多子零件组成，轮胎是单独的类，引擎是单独的类，车体也是单独的类。这些类组合在一起构成更加强大的汽车类。

JavaScript 语言中虽然没有“类”结构，但是通过原型链，我们可以很容易地实现上面所提到的面向对象特性。

6. 多态

不同对象对同一消息有不同的响应，就是多态。举例而言，上课铃一响，所有教师都要开始教学动作，但是不同科目的教程所教的内容一定不同。上课铃就可以理解为消息，其让教师执行教学动作，教学动作就是对象的响应，不同对象响应不同。

3.4.1　JavaScript 中模拟类的方式

JavaScript 当初在设计时只是为了执行一些简单的浏览器脚本，如今 JavaScript 几乎无所不能，从前端到后端，使用 JavaScript 完成的项目越来越庞大。不幸的是，JavaScript 本身并不支持类，

也就是说 JavaScript 中并没有类的概念。如果我们深入挖掘一下类的定义，就会发现在 JavaScript 中模拟出类真的是十分灵活和简单。

类说白了就是描述一类对象的通用属性，也可以理解为类是对象的模板。如果我们需要一个教师对象，我们可以直接定义这个教师对象。如果我们需要很多教师对象，你可能已经想到了：我们可以定义一个工厂方法来生成教师对象。这样，在需要教师对象的时候，不需要再重新定义，调用这个工厂方法生成一个教师实例对象即可。这个工厂方法实际上就起到了类的作用，在 JavaScript 中，这种函数被称为构造函数。

博 闻 强 识

在很多面向对象语言中，类是对象的基础，对象是由类构造出来的。一般，类名首字母大写，对象名首字母小写。这是一种编程规范。

JavaScript 中模拟类的方法有很多种，如果有兴趣，也可以自己创造一种。这里将介绍 4 种模拟类的方法来抛砖引玉，希望可以帮助大家打开更宽广的思路。

1. 使用工厂方法模拟类

定义一个函数，在函数内部创建一个空对象，并对它进行一些类实例的完善，示例如下：

```
//模拟类
function Teacher(name,age,subject){
    var obj = {};
    obj.name = name;
    obj.age = age;
    obj.subject = subject;
    obj.teaching = function(){
        console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
    }
    return obj;
}
var jaki = Teacher("Jaki","25","JavaScript");
var lucy = Teacher("Lucy","24","Swift");
jaki.teaching();//我是 Jaki,欢迎大家来听 JavaScript 教学课程。
lucy.teaching();//我是 Lucy,欢迎大家来听 Swift 教学课程。
```

2. 构造方法模拟类

使用构造方法模拟创建一个教师类，示例如下：

```
//模拟类
function Teacher(name,age,subject){
    this.name = name;
    this.age = age;
    this.subject = subject;
    this.teaching = function(){
    console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
```

```
    }
}
var jaki = new Teacher("Jaki","25","JavaScript");
var lucy = new Teacher("Lucy","24","Swift");
jaki.teaching();//我是 Jaki,欢迎大家来听 JavaScript 教学课程。
lucy.teaching();//我是 Lucy,欢迎大家来听 Swift 教学课程。
```

new 关键字我们前面介绍过，当使用 new 关键字调用一个函数时，首先创建一个空对象，将这个对象与构造方法中的 this 进行绑定并建立原型链，之后执行构造方法进行对象的构造，最后将此对象返回。

3. 使用 Object 构造方法对象的 create()方法模拟类

在前面章节已经学习了 Object 构造方法对象中的 create()函数，这个函数的作用是创建一个以某个对象为原型的对象。我们可以使用这个方法来模拟类，示例如下：

```
var Teacher = {
    name:"Jaki",
    age:25,
    subject:"JavaScript",
    teaching:function(){
        console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
    }
}
var jaki = Object.create(Teacher);
jaki.teaching();//我是 Jaki,欢迎大家来听 JavaScript 教学课程。
```

使用这种方式来模拟类有一个弊端，在构造实例对象的时候没有办法传入参数，如果要对对象的属性进行赋值，必须在构造对象后再单独进行设置。

4. 封装法

封装法模拟类的思路来源于其他面向对象语言的启发。示例如下：

```
var Teacher = {
    init:function(name,age,subject){
        var teacher = {};
        teacher.name = name;
        teacher.age = age;
        teacher.subject = subject;
        teacher.teaching = function(){
          console.log("我是"+this.name+",欢迎大家来听"+this.subject+"教学课程。");
        }
        return teacher;
    }
};
var jaki = Teacher.init("Jaki",25,"JavaScript");
var lucy = Teacher.init("Lucy",24,"Swift");
```

```
jaki.teaching();//我是Jaki,欢迎大家来听JavaScript教学课程。
lucy.teaching();//我是Lucy,欢迎大家来听Swift教学课程。
```

使用封装法模拟类有一些先天的优势，可以很方便地在类中定义私有属性和私有方法，也可以区别于实例方法和属性很容易地定义类方法和类属性。

博 闻 强 识

在许多面向对象语言中都有实例属性、实例方法与类属性、类方法的定义，有些语言也会用静态属性和静态方法来描述类属性类方法（如Swift）。实例属性实例方法是绑定在实例对象上的，类属性与类方法则是直接绑定在类上的。

3.4.2 在 JavaScript 中实现继承机制

在 JavaScript 中实现继承机制的方式也多种多样。和模拟类的学习相似，这里将介绍 3 中实现继承机制的方法，分别为对象冒充法、原型法和混合法。希望学习了这些方法后可以打开思维，更深入地理解 JavaScript 语言的精髓。

1. 使用对象冒充的方式实现继承

对象冒充的方式利用了 JavaScript 中 this 关键字的特性（this 关键字在函数内出现时，表示调用此函数的对象）。我们还以教师类为例，首先所有的教师都是人类，人类都有年龄和姓名，因此，我们可以创建一个 People 类作为教师类的父类，示例代码如下：

```
//创建People类作为父类
function People(name,age){
    this.name = name;
    this.age = age;
}
function Teacher(name,age,subject){
   //这一步的作用是转换this的指向
    this.init = People;
    this.init(name,age);
    delete this.init;
   //添加教师类特有的属性
    this.subject = subject;
    this.teaching = function(){
        console.log("教师"+this.name+"正在教授"+this.subject+"课程。");
    };
}
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.teaching();//教师Jaki正在教授JavaScript课程。
lucy.teaching();//教师Lucy正在教授Swift课程。
```

上面的示例代码创建了一个最简单的继承体系，Teacher 实例对象成功继承了 People 类中定义

的姓名和年龄属性。其实可以将上面的代码进行简化，改变函数内 this 的指向，使用 call 函数和 apply 函数都可以做到，示例如下：

```
function People(name,age){
    this.name = name;
    this.age = age;
}
function Teacher(name,age,subject){
    People.call(this,name,age);
    this.subject = subject;
    this.teaching = function(){
        console.log("教师"+this.name+"正在教授"+this.subject+"课程。");
    };
}
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.teaching();//教师 Jaki 正在教授 JavaScript 课程。
lucy.teaching();//教师 Lucy 正在教授 Swift 课程。
```

要实现多继承也十分容易，示例如下：

```
function People(name,age){
    this.name = name;
    this.age = age;
}
function Work(time){
    this.time = time;
}
function Teacher(name,age,subject){
    People.call(this,name,age);
    Work.call(this,8);
    this.subject = subject;
    this.teaching = function(){
        console.log("教师"+this.name+"正在教授"+this.subject+"课程。"+"工作时间：
"+this.time+"小时。");
    };
}
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.teaching();//教师 Jaki 正在教授 JavaScript 课程。工作时间：8 小时。
lucy.teaching();//教师 Lucy 正在教授 Swift 课程。工作时间：8 小时。
```

上面代码创建了一个人类和一个职业类，教师类既有人类的属性也有职业类的属性。

使用对象冒充的方式实现继承体系十分简单，但是严格上讲，其并非是真正意义上的继承，使用原型链的方式会实现更加类似于传统意义上的继承。

博 闻 强 识

多继承是指一个子类可以有多个父类。C++是支持多继承的一种语言，Java、Objective-C、Swift 等语言并不支持这种特性。

2. 使用原型链的方式实现继承

原型链是 JavaScript 中最难理解也是最令人费解的一部分。为了便于理解，我们可以再来研究一下 new 关键字到底做了什么。首先，在 JavaScript 中，你要始终保持除原始值外万事万物都是对象这样一种思想，函数也是一种对象。当创建函数时，首先会创建这个函数对象本身，除此之外，还会创建一个对象作为此函数对象的 prototype 属性，例如：

```
function People(){
    this.sayHi=function(){
        console.log("Hello,I am "+this.name+","+this.age+" years old。");
    }
}
console.log(People.prototype);//People {}
```

你一定奇怪，这个 People 函数的 prototype 为什么会打印出 People {}信息。其实这只是一种格式化的输出，函数的 prototype 就是一个普通的 Object 对象，这个对象中自动生成了一个属性 constructor，constructor 默认指向当前的函数对象，即 People。当使用 new 关键字调用 People 方法进行实例对象的构造时，会以如下步骤进行：

步骤 01 创建空对象{}，暂且命名为 obj。

步骤 02 将构造函数中的 this 指向 obj，并将 obj 的__proto__属性指向构造函数的 prototype 属性，建立原型链。

步骤 03 执行构造函数。

步骤 04 将对象 obj 返回。

这里还有必要介绍一些__proto__属性，其是 Object 实例对象中的一个私有属性，这个属性指向了当前对象的原型对象。

上面的文字描述可能不是很容易理解，图 3-1 比较直观地解释了原型链的原理。

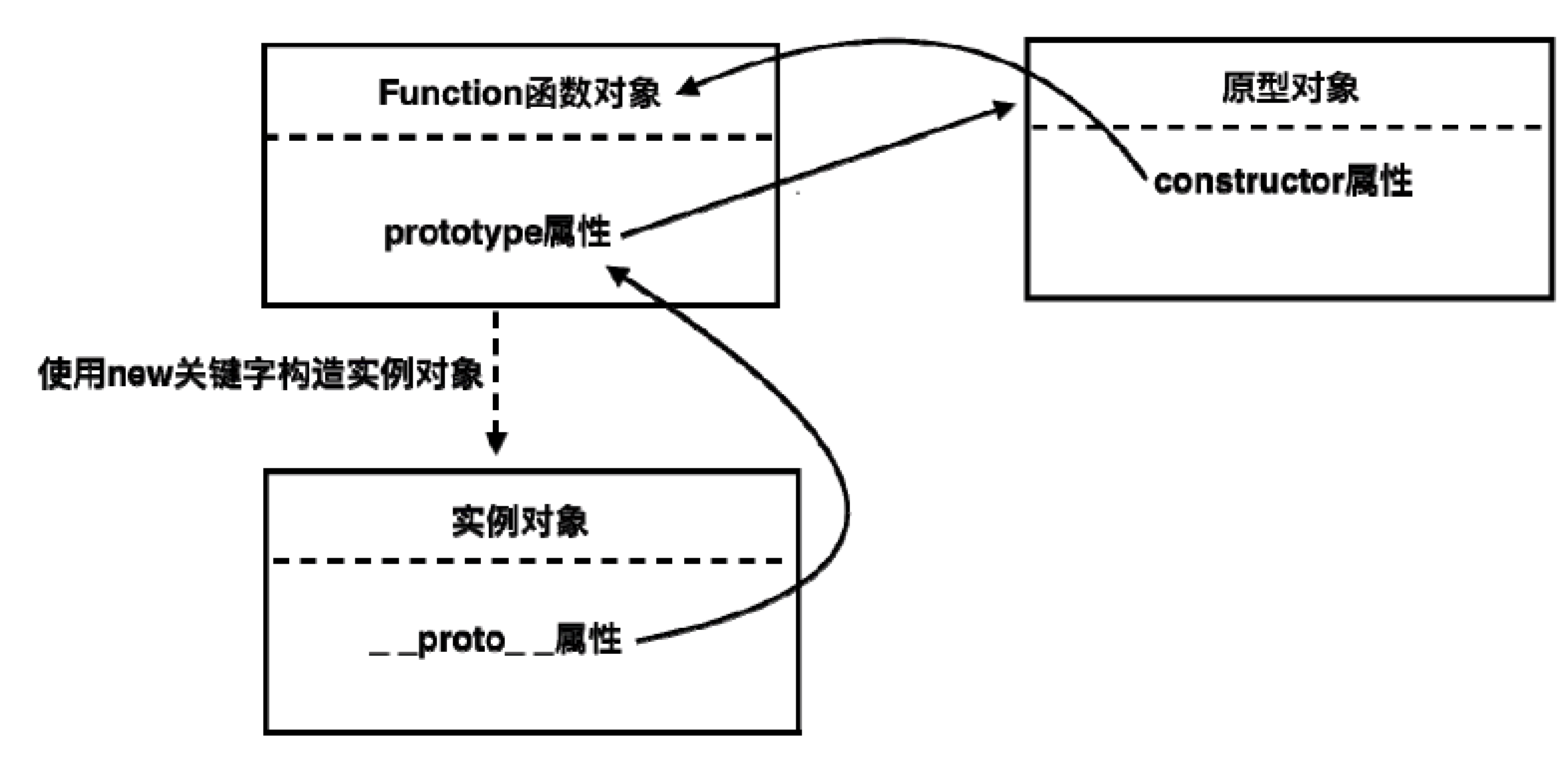


图 3-1 原型链示意图

使用原型链的方式实现继承，示例代码如下：

```
function People(){
    this.sayHi=function(){
        console.log("Hello,I am "+this.name+","+this.age+" years old。");
    }
}
function Teacher(name,age,subject){
    this.name = name;
    this.subject = subject;
    this.age = age;
}
//设置 prototype 属性
Teacher.prototype = new People();
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.sayHi();//Hello,I am Jaki,25 years old。
lucy.sayHi();//Hello,I am Lucy,24 years old。
```

如上代码所示，在 Teacher 实例对象中并没有 sayHi 方法，在代码执行时，首先会在 Teacher 实例对象内部寻找这个方法，如果没有找到，会向实例对象的__proto__属性中去找，这个属性其实就是 Teacher 函数对象的 prototype 指向的对象，即 People 实例对象。在这个 People 实例对象中找到了 sayHi 方法，将此方法返回，如果没有找到，会再向 People 实例对象的__proto__属性里找，一层一层递归下去，直到找到或者最终对象的__proto__属性为 undefined。访问属性时也是一样的原理，这便形成了一套完整的原型链。

使用原型链实现的继承有一个很大的优势，其可以使用 instanceof 关键字来检查某个构造函数是否在对象的原型链上，这十分类似于检查某个对象是否为某个类或其子类的实例，示例如下：

```
console.log(jaki instanceof People);//true
```

当然原型链的继承模式也有一些缺点，比如原型链是单链，无法实现多继承。通过原型链继承来的子类的实例在访问属性时，可能会付出一定的性能做代价，因为其总是递归遍历从下往上查找属性的，如果访问一个 undefined 的属性，此性能代价会更大。

抽 丝 剥 茧

在 JavaScript 中没有严格的私有属性的定义，这里说的私有属性其实是编码习惯上的，在开发中，我们常常把不想让外界访问的属性使用双下画线开头和结尾，将其作为一种编码规范上的私有。

3. 混合模式

对象冒充的方式实现的继承归根到底是一种假继承，实质上所有实例对象都有完整的一套属性和方法，一般不同对象的属性值各有差异，每个对象都有一套独立的属性，这点倒不是问题，问题在于所有同类的实例对象方法都是一致的，每个对象都有独立的一套就失去了继承的意义。原型链的方式实现继承时，父类的方法是放在原型对象中的，并不在对象本身里，因此所有同类的实例

对象都是共享这些方法的，但是其也有一个致命的缺陷，这种方法无法在原型对象创建时指定参数，这也是为什么我们上面的示例中将教师姓名、年龄属性放在了子类中而不是父类。那么如何实现继承才是最优的模式呢？答案很简单，将属性使用对象冒充的方式来继承，方法使用原型链的方式来继承。示例如下：

```
function People(name,age){
    this.age = age;
    this.name = name;
}
//方法都放入原型链中
People.prototype = {
    constructor:People,
    sayHi:function(){
        console.log("Hello,I am "+this.name+","+this.age+" years old。");
    }
}
function Teacher(name,age,subject){
    //用对象冒充把属性继承过来
    People.call(this,name,age);
    this.subject = subject;
}
Teacher.prototype = new People();
var jaki = new Teacher("Jaki",25,"JavaScript");
var lucy = new Teacher("Lucy",24,"Swift");
jaki.sayHi();//Hello,I am Jaki,25 years old。
lucy.sayHi();//Hello,I am Lucy,24 years old。
```

这种混合模式实现的继承也是基于原型链的，因此也可以使用 instanceof 关键字判断对象是否是某个类或其子类的实例。

博 闻 强 识

许多编译型语言对语法都有着严格的控制，相比之下，JavaScript 真的是一种非常灵活的语言，只要你敢想，动手去尝试，就会发现其中乐趣无限。

第 4 章

ECMAScript 6 新特性

前面的章节我们一直在使用 JavaScript 这个名词，ECMAScript 又是什么呢？它和 JavaScript 之间有什么关系？要明白这个问题，首先需要了解两个概念：标准与实现。

标准也可以理解为一套规则，任何人造的结构化的事物都需要有一套标准，例如工业中生产的螺母，生产螺母的厂家可能各种各样，但是螺母的工业标准是一致的。ECMAScript 就是一种标准，其是浏览器脚本语言的标准，JavaScript 语言就是 ECMAScript 标准的一种实现，当然实现 ECMAScript 标准的浏览器脚本语言不只有 JavaScript 一种（例如 Jscript 和 ActionScript），但 JavaScript 无疑是其中最完善最流行的一种。ECMAScript 6（以下简称 ES6）是指 ECMAScript 标准的第六版，由于其和上一版相比差别很大，并且使用 React Native 开发移动应用时使用的也是 ES6 的标准，因此我们有必要对 ES6 中的新特性进行系统的学习。

抽 丝 剥 茧
你前面使用的JavaScript解释程序是支持ES6语法的，只是我们还没有写新特性相关的代码。

4.1　ECMAScript 6 的块级作用域

在前面介绍 JavaScript 语法的内容中，并没有提到作用域这个概念。原因是 JavaScript 中有一种十分特殊的语法规则：变量提升。如果你使用了一个从未声明过的变量，程序运行会直接抛出异常。但是如果代码中有过对此变量的声明，无论在声明前使用还是在声明后使用，程序都不会再抛出异常，例如：

```
//变量提升
console.log(name);//undefined
```

```
var name = "Jaki";
console.log(name);//Jaki
```

造成这种语法特性的原因是 JavaScript 解释器在对代码进行扫描时，会将声明的变量和函数先定义为全局符号，运行到具体定义处时才进行赋值。这种语法特性多多少少会造成一些误解，在许多其他流行编程语言中，变量在声明之前是严格不能使用的。如果说上面的示例代码造成误解的问题不大，那么下面的代码就更能说明问题了：

```
//变量提升
console.log(name);//undefined
if (false) {
    var name = "Jaki";
}
console.log(name);//Jaki
for (var i = 0 ; i < 3; i++){
    var sum = i;
}
console.log(i);//3
console.log(sum);//2
```

在上面的示例代码中，首先 if 条件语句使用为假，if 代码块永远不会执行到，但是从打印结果可以看出 name 变量还是被声明了，这已经有些奇怪了，对吧？后面的循环结构中，我们在 for 循环条件结构中声明了一个 i 变量，在循环体内声明了 sum 变量，可是当循环结束后，i 变量依然存在，变成了全局变量。sum 变量同样也变成了全局变量，这就不仅是奇怪的问题了，这种变量提升会消耗一部分无用内存并对之后代码的编写产生额外的风险。在 ES6 标准中，新引入了块级作用域，可以完美地解决这些问题。

4.1.1 let 关键字

ES6 标准中的块级作用域实际上是由 let 与 const 关键字决定的。在 ES6 标准中新引入了 let 和 const 关键字，其用法十分类似于 var 关键字，都是进行变量的声明。不同的是，var 声明的变量会存在变量提升和泄露为全局变量的问题，let 和 const 声明的变量则只在其所在的代码块中有效。举例如下：

```
//块级作用域
{
    var a = 10;
    let b = 10;
    console.log(b);//10
}
console.log(a);//10
console.log(b);//程序抛出异常
```

上面的代码当程序运行到 console.log(b)时会抛出异常。也就是说，使用 let 命令声明的变量，

一旦脱离其所在的代码块，这个变量就不再可以使用。这种局部变量十分适合用于 for 循环结构中，例如：

```
for(let i=0;i<3;i++){
}
console.log(i);//程序抛出异常
```

let 命令还有一个规则，其不存在变量提升，即在变量声明之前，此变量是不可使用的，而不是 undefined，例如：

```
console.log(a);//程序抛出异常
let a = 10;
```

使用 let 声明的变量也不可以进行重复声明，如下的写法也会抛出异常：

```
let a = 10;
let a = 11;//程序抛出异常
```

在 ES6 标准中，还存在一个暂时性死区的概念。只要块级作用域中使用 let 命令进行了变量声明，那么它所声明的变量就会绑定进这个区域不再受外部影响。这种语法特性就称为暂时性死区，示例如下：

```
let tmp = 10;
{
    console.log(tmp);//程序抛出异常
    let tmp = 11;
    console.log(tmp);//11
}
```

上面的示例代码说明，在块级作用域中使用 let 声明了变量，那么在声明之前，这个变量都不能进行使用（尽管全局中也定义了这样一个变量）。

我们回过头再来理解下块级作用域。在 ES5 标准中除了函数和 try-catch 会生成作用域外，没有额外的块级作用域这个概念的，我们在编写代码时很容易产生内层变量覆盖外层变量和局部变量泄露为全局变量的问题。块级作用域做到了作用域内的变量不受外界影响同时也不会影响外界，提高了代码的安全性。块级作用域也是可以嵌套的，外层作用域无法读取内层作用域的变量，示例如下：

```
//将打印
/*
Hello World
Wa
New
*/
{
    let a = "New";
    {
        let a = "Wa";
```

```
        {
            let a = "Hello World";
            console.log(a);
        }
        console.log(a);
    }
    console.log(a);
}
```

对于函数的声明，在 ES6 中遵守块级作用域的规则，在作用域内声明的函数只能在作用域内使用，例如：

```
{
    let func = function(){
        console.log("function");
    }
    func();//function
}
func();//程序抛出异常
```

4.1.2 const 关键字

在许多编程语言中除了有变量的概念外，还有常量的概念。常量就是值不能改变的量，在 ES6 标准中，使用 const 关键字来进行常量的声明。修改常量的值会使程序抛出异常，示例如下：

```
const PI = 3.14;
PI = 3;//抛出异常
```

需要注意，在使用 const 声明变量时要同时为其赋值，一旦 const 变量被定义，后面就不能够再对它进行修改。const 关键字声明的变量和 let 关键字声明的变量享有同样的作用域规则，这里就不再赘述。

const 声明的常量有一点需要额外注意，const 实际保证的是常量空间所存储的数据不能修改，并不一定是常量所对应的值不能修改。如果常量对应的是一个对象，你可以对此对象的属性方法等进行修改，但是不可以直接将此常量指向的对象修改掉，示例如下：

```
const teacher = {
    name:"Jaki",
    age:25
};
//对对象进行修改没问题
teacher.name = "Lucy";
teacher.age = 24;
//直接修改常量的指向则会报错
teacher = {
```

```
    name:"Lucy",
    age:24
};
```

4.2　解构赋值

解构赋值是 ES6 中一个十分强大的特性。掌握了这种技术，你将能十分轻松地从数组、对象等结构中提取所需要的值。解构赋值通俗理解就是分解数据的结构，为变量进行赋值。这种语法特性广泛应用于数组的分解、对象的分解、字符串的分解、函数参数的分解等。

4.2.1　数组的解构赋值

通过前面知识的学习，你应该已经熟练掌握了 JavaScript 中数组结构的用法。你一定还记得，数组实际上是一种特殊的对象，其属性名是递增的整数。如果我们要从数组中取值，通常会采用如下方式：

```
let students = ["Jaki","Lucy","Mery","July"];
//取数组中第 1 个元素
let stu1 = students[0];//Jaki
//取数组中第 2 个元素
let stu2 = students["1"];//Lucy
```

如果我们需要把数组中所有的值都提取到对应的变量中，那么上面的方法是十分麻烦的，需要手动地声明每个变量，并对它进行赋值。在 ES6 中使用解构赋值，问题将变得十分简单，示例如下：

```
//进行数组的解构赋值
let [a,b,c,d] = students;
console.log(a+" "+b+" "+c+" "+d);//Jaki Lucy Mery July
```

上面代码将数组中的值按顺序提取到了 a、b、c、d 变量中。需要注意，解构赋值前面的 let 关键字和正常的变量声明意义一致，因此如果解构赋值中新声明的变量在前面声明过，则程序会抛出异常（当然使用 var 关键字不会有这个问题）。上面示例的代码 students 数组中有 4 个元素，进行解构赋值的表达式中也声明了 4 个变量，数组中的元素刚好可以和解构赋值表达式中的变量一一对应，这种解构赋值的场景叫做完全解构。与其相对，如果解构赋值表达式中声明的变量与数组中的元素个数并不一一对应，就会产生不完全解构，示例如下：

```
//不完全解构
//只提取数组中的前三个数据
let [e,f,g] = students;
console.log(e+" "+f+" "+g);//Jaki Lucy Mery
//只提取数组中的第四个数据
let [,,,h] = students;
```

```
console.log(h);//July
//提取数组中的第 1 个值，并将余值放入另一个数组
let [i,...j] = students;
console.log(i+" "+j);//Jaki [Lucy,Mery,July]
//溢出的变量将被赋值为 undefined
let [k,l,m,n,o] = students;
console.log(k+" "+l+" "+m+" "+n+" "+o);//Jaki Lucy Mery July undefined
```

数组的解构赋值也支持嵌套，只要解构表达式的嵌套结构与数组的嵌套结构一致，就可以解构赋值成功，例如：

```
//解构赋值的嵌套
let array = [1,2,[5,6,7]];
let [p,q,[r,s,t]] = array;
console.log(""+p+q+r+s+t);//12567
```

如果解构表达式声明的变量有溢出，则会解构失败，解构失败的变量将默认被赋值为 undefined。在解构表达式中，你也可以为变量设置一个默认值，当解构失败或者解构出的值为 undefined 时，此变量会采用设置的默认值作为自己的值，示例如下：

```
//设置默认值
let [u=0,v=0,w=0] = [1,undefined];
console.log(u+" "+v+" "+w);//1 0 0
```

需要注意，必须解构失败或者解构出的值严格为 undefined 时变量才会采用默认值，其他解构出的 null、false、NaN 等都不会触发变量默认值。例如：

```
[u=0,v=0,w=0] = [1,NaN,null];
console.log(u+" "+v+" "+w);//1 NaN null
```

如上代码所示，其实在进行解构赋值时，并一定要声明新的变量，也可以将数组中的数据解构赋值到已经存在的变量中。需要注意，如果是对象的解构赋值，则必须将解构赋值表达式放入小括号中，后面会具体介绍。

4.2.2 对象的解构赋值

和数组的解构赋值类似，对象也可以进行解构赋值。使用对象的解构赋值可以方便地提取对象的属性。例如：

```
//对象的解构赋值
let teacher = {
    name:"Jaki",
    age:25,
    teaching:function(){
        console.log("teaching...");
    }
};
```

```
let {name,age,teaching} = teacher;
console.log(name+" "+age); //jaki 25
teaching();//teaching...
```

对象中的属性是没有先后顺序之分的，因此在对对象进行解构赋值时，赋值的变量名必须和对象中的属性名完全一致，否则会解构失败。如果要自定义解构赋值的变量名，可以采用如下映射方式：

```
let {name:myName,age:myAge} = teacher;
console.log(myName+" "+myAge); //jaki 25
```

博 闻 强 识

其实对象的结构赋值的默认结构是这样的：

```
let {name:name,age:age} = teacher;
```

当要解构赋值的变量名与对象的属性名相同时，可以省略映射结构。数组也是对象，因此也可以使用如下方式对数组进行解构：

```
let {0:x,1:y} = [1,2];
console.log(x+" "+y);//1 2
```

对象的解构赋值也是支持嵌套的，这在处理复杂对象时十分高效，例如：

```
let teacher = {
    name:"Jaki",
    age:25,
    students:[
        {
            name:"Lucy",
            age:24
        },
        {
            name:"July",
            age:26
        }
    ],
    teaching:function(){
        console.log("teaching...");
    }
};
let {students:[{name:name1},{name:name2}]} = teacher;
console.log(name1+" "+name2);//Lucy July
```

如果是将对象解构赋值到已有的变量中，则解构赋值表达式必须放在小括号内，否则会抛出异常，原因是 JavaScript 解释器会将大括号解析为代码块而不是表达式，例如下面的写法是正确的：

```
({name:name,age:age} = teacher);
```

在对象解构赋值时，也可以为变量添加默认值，如前面所说，只有当解构失败或者解构出的值为严格的 undefined 时才会触发默认值。

4.2.3 字符串与函数参数的解构赋值

在进行字符串的解构赋值时，可以将字符串理解为一个字符数组，示例如下：

```
let [c1,c2,c3,c4,c5] = "Hello";
console.log(c1+c2+c3+c4+c5);//Hello
```

这种解构赋值常用于提取字符串中某个位置的字符。在 ES6 中，函数的参数也可以进行解构赋值，在 ES5 标准中，如果你需要向函数中传递数组或对象，通常要这样做：

```
let people = {
    name:"Jaki",
    age:25
};
function print(people){
    console.log(people.name+":"+people.age);
};
print(people);//Jaki:25
```

使用解构赋值的方式，我们可以这样写：

```
let people = {
    name:"Jaki",
    age:25
};
function print({name,age}){
    console.log(name+":"+age);
};
print(people);//Jaki:25
```

同样你也可以为函数的参数在解构赋值时设置一个默认值。需要特别注意，为函数参数解构赋值设置默认值和设置函数参数的默认值是完全不同的，请看如下示例：

```
function print({name="name",age=0}={name:"Jaki",age:25}){
    console.log(name+":"+age);
};
print({});//name:0
print();//Jaki:25
```

上面的示例代码中，如果解构赋值失败或者解构为 undefined，参数 name 会采用默认值“name”，参数 age 会采用默认值 0。但是如果调用函数时不传入参数，则会采用参数对象的默认值，即{name:"jaki",age:25}这个对象，为函数的参数设置默认值也是 ES6 的特性之一。

博 闻 强 识

我们来玩一个小游戏，交换两个变量的值，最少需要几步？你可能会不假思索地写出如下代码:

```
let v1=10;
let v2=11;
let v3 = v1;
v1 = v2;
v2 = v3;
console.log(v1);
console.log(v2);
```

上面的代码需要创建中间变量，并且需要至少 3 步才能完成两个变量值的交换，如果使用解构赋值技术，不仅可以省略中间变量，而且一步即可完成:

```
let v1=10;
let v2=11;
[v1,v2] = [v2,v1];
console.log(v1);
console.log(v2);
```

4.3　箭头函数

箭头函数是 ES6 新引入的一种创建函数的语法规则，这种语法可以大程度地简化函数编写的格式，并且会对函数中的 this 进行绑定。

4.3.1　箭头函数的基本用法

前面我们介绍过好几种在 JavaScript 中创建函数的方法，虽然有所差别，但其在格式上都大同小异，用法上也基本一致。例如：

```
function exp(a){
    return a*a;
}
let res = exp(5);
console.log(res);
```

如果使用箭头函数语法对上面的函数进行重写，结果如下：

```
let f = (a)=>{
    return a*a;
}
```

```
let res = f(5);
console.log(res);
```

箭头函数的基本语法格式为：(参数列表)=>{函数体}。如果箭头函数只有一个参数，并且函数体中只有一行代码，我们可以再进行简化，示例如下：

```
let f = a=>a*a;
let res = f(5);
console.log(res);
```

看到如此简洁的函数定义，是不是有眼前一亮的感觉呢？

抽 丝 剥 茧

如果箭头函数的函数体只有一句代码，并且返回的是一个对象，需要将对象包裹在小括号内，这是因为大括号默认会被解释为代码块，示例如下：

```
let func = ()=>({name:"Jaki"});
```

箭头函数在用法上和我们前边介绍的普通函数并没有太大的差异，同样也支持函数参数的解构赋值，示例如下：

```
let func = ({name,age})=>console.log(name,age);
func({name:"Jaki",age:25});//Jaki 25
```

箭头函数这种简洁的结构十分适用于回调函数的编写，在后面的开发中，我们也会经常使用箭头函数。

4.3.2 箭头函数中 this 的固化

箭头函数有一个十分重要的特点，其中 this 是被固化的。普通函数中 this 默认是指向调用函数的对象，当然也可以使用 call、apply、bind 这些函数进行 this 指向的更改。但是在箭头函数中，this 是被固化的，也就是说其中的 this 在定义函数时就已经被绑定，不能修改，也和调用者无关。比较下面两段代码。

普通函数：

```
let teacher = {
    name:"Jaki",
    age:25,
    print:function(){
        console.log(this.name,this.age);
    }
}
let student = {
    name:"Lucy",
    age:24,
    print:teacher.print
```

```
}
teacher.print();//Jaki 25
student.print();//Lucy 24
```

箭头函数：

```
//箭头函数 this 的固化
let teacher = {
    name:"Jaki",
    age:25,
    print:()=>{
        console.log(this.name,this.age);
    }
}
let student = {
    name:"Lucy",
    age:24,
    print:teacher.print
}
teacher.print();//undefined undefined
student.print();//undefined undefined
```

可以发现，箭头函数中的 this 实际上并非指向 teacher 对象或者 student 对象，而是指向了全局对象，即箭头函数内部的 this 就是定义时所在环境的 this 指向，并且会被固化，不能修改。再来看下面这个例子：

```
function foo(){
    this.name = "foo";
    this.inline = ()=>{
        console.log(this.name);
    };
    this.outline = function(){
        console.log(this.name);
    }
}
let obj = new foo();
obj.inline();//foo
obj.inline.call({name:"Jaki"});//foo
obj.outline();//foo
obj.outline.call({name:"Jaki"});//Jaki
```

从上面的代码可以看出，箭头函数在定义时 this 就固化成为 foo 函数对象本身，无论调用方如何修改，这个 this 指向都不变。为了简单理解，我们可以将箭头函数中的 this 解析如下：

```
function foo(){
    let _this = this;
    this.name = "foo";
```

```
        this.inline = ()=>{
            console.log(_this.name);
        };
        this.outline = function(){
            console.log(this.name);
        }
    }
```

由于箭头函数 this 固化的特性，因此不能将箭头函数作为构造函数来使用，即不可以使用 new 关键字来调用箭头函数。

抽 丝 剥 茧

定义的全局对象中箭头函数中的 this 之所以指向全局对象，是因为对象可以理解为是全局对象调用 Object 构造函数创建的，这个函数中的 this 当然指向其调用方全局对象，因此箭头函数中的 this 固化成了全局对象。

4.4 Set 与 Map 数据结构

在本书前边的内容中，我们学习了 Array 这种数据结构，抛开其是对象的本质不说，你可以将 Array 看成是一种有序的集合结构，其中的元素因为是有顺序的，因此也可以重复。Set 是 ES6 标准中新引入的一种集合数据结构，其中元素无下标（但是可以有序地进行遍历），也不可以重复。Map 也是 ES6 中新引入的一种数据结构，十分类似于对象，但是也有不同，本节我们会具体介绍这两种数据结构。

4.4.1 Set 集合结构

Set 是一种集合结构，允许你存储任意类型数据的唯一值。所谓唯一值，是指所存储的值不能重复。构造一个 Set 集合对象的示例如下：

```
//创建 Set 集合
let set = new Set([1,2,3,4,4,2]);
console.log(set);//Set { 1, 2, 3, 4 }
```

Set 构造函数中可以传入一个数组对象，数组中的值会被作为 Set 集合的元素插入集合中。需要注意，数组中的元素如果有重复，就会自动被剔除，最终生成的 Set 集合中的元素都是唯一的。Set 集合的这种特性也可以作为一种数组去重的好方法。Set 实例对象的 size 属性可以获取到集合中元素的个数，示例如下：

```
let set = new Set([1,2,3,4,4,2]);
console.log(set.size);//4
```

关于 Set 实例对象中元素的操作，可以使用如下示例方法：

```
let set = new Set();
//向集合中插入元素
set.add("Jaki");
set.add("Lucy");
console.log(set);//Set { 'Jaki','Lucy' }
//删除集合中的某个元素
set.delete("Jaki");
console.log(set);//Set {'Lucy'}
//删除集合中的所有元素
set.clear();
console.log(set);//Set {}
set.add("Jaki");
set.add("Lucy");
//返回 Set 集合迭代器对象
console.log(set.entries());//SetIterator { [ 'Jaki', 'Jaki' ], [ 'Lucy',
'Lucy' ] }
//让集合中的所有元素调用一次回调方法
/*
将打印
Jaki
Lucy
*/
set.forEach((element)=>{
    console.log(element);
},set);
//判断集合中是否包含某个元素
console.log(set.has("Jaki"));//true
//下面这两个方法都是用来获取集合中所有元素的迭代器
console.log(set.keys());//SetIterator { 'Jaki', 'Lucy' }
console.log(set.values());//SetIterator { 'Jaki', 'Lucy' }
```

上面代码中的注释十分详尽，Set 实例对象中的 forEach 方法和 Array 实例对象的 forEach 方法行为基本一致，都是将其中元素遍历一遍，对每个元素执行传入的回调方法，forEach 函数的第 2 个参数传入的对象会被绑定到回调函数中的 this 上。Set 集合可以使用 for-of 结构进行迭代，示例如下：

```
/*
Jaki
Lucy
*/
for(item of set){
    console.log(item);
}
```

抽 丝 剥 茧

对象属性的遍历使用的是 for-in 结构，Set 元素的遍历使用的是 for-of 结构，切记不要混淆。

ES6 中还定义了一种特殊的 Set 集合：WeakSet。正如其名，它是一种弱引用集合。与 Set 集合相比，它有两个特点：

（1）WeakSet 中只能存放引用数据，即对象数据，不能存放原始值。
（2）WeakSet 中的对象元素都是弱引用的，因此其无法进行枚举。

WeakSet 实例对象中提供的方法如下：

```
let obj1 = {name:"Jaki"};
let obj2 = {name:"Lucy"};
let wSet = new WeakSet([obj1]);
//弱引用集合中是否包含某个元素
console.log(wSet.has(obj1));//true
//添加一个元素
wSet.add(obj2);
//删除一个元素
wSet.delete(obj1);
```

抽 丝 剥 茧

弱引用是指，除了集合之外，若没有其他变量或属性引用这个对象，则这个对象值会被垃圾回收机制回收掉，集合中的这个元素也将无效。

4.4.2 Map 字典结构

你一定有过查字典的经历。以汉语字典为例，当你需要查找某个字的释义时，先要在字典的索引中找到这个字，然后根据索引提供的页标来找到这个字的释义。在字典结构中，我们常常把这个“索引字”称为键，把“释义”称为值。

许多编程语言中都有字典数据结构，在 ES6 标准前，JavaScript 中的对象可以充当字典来使用，但是对象的属性不能是任意类型的值，ES6 标准中引入了 Map 数据结构。Map 就是简单的键值映射，其中键和值都可以是任意值。和 Set 数据结构相似的是，Map 中的键也都是唯一的（不同键的值可以相同）。

Map 实例对象中的 size 属性可以获取到其中键值对的个数（去重后），示例如下：

```
//Map 字典
let map = new Map([["name","Jaki"],["age",25],[321,true],[321,true]]);
console.log(map.size);//3
```

下面列出一些常用的操作 Map 实例对象的方法：

```
let map = new Map();
//向 Map 实例对象中添加键值对
```

```
map.set("name","Jaki");
map.set("age",25);
map.set(123,true);
console.log(map);//Map { 'name' => 'Jaki', 'age' => 25, 123 => true }
//删除一对键值
map.delete(123);
console.log(map);//Map { 'name' => 'Jaki', 'age' => 25 }
//返回一个 Map 迭代对象
console.log(map.entries());//MapIterator { [ 'name', 'Jaki' ], [ 'age', 25 ] }
//判断 Map 实例对象中是否包含某个键
console.log(map.has("name"));//true
//获取 Map 中某个键的值，如果键不存在，就会返回 undefined
console.log(map.get("name"));//Jaki
//获取 Map 中的所有键
console.log(map.keys());//MapIterator { 'name', 'age' }
//获取 Map 中的所有值
console.log(map.values());//MapIterator { 'Jaki', 25 }
/*
将打印
name Jaki
age 25
*/
map.forEach((value,key)=>{
    console.log(key,value);
},map);
//清空 Map 中的所有键值
map.clear();
console.log(map);
```

Map 实例对象调用 forEach 方法来对自身进行遍历，并对每个键值对执行回调方法，回调函数中第 1 个参数为当前键值对的键，第 2 个参数为当前键值对的值。

Map 数据结构也可以使用 for-of 结构来进行遍历，示例如下：

```
/*
name Jaki
age 25
*/
for(let [a,b] of map){
    console.log(a,b);
}
```

与 WeakSet 相对应，ES6 中也定义了一个 WeakMap。WeakMap 是一种特殊的 Map，其中所有的键只能是引用类型（对象），值可以是任意类型，并且其对所有对象键的引用都是弱引用，因此 WeakMap 也是不可以枚举的。WeakMap 实例对象可用方法列举如下：

```
let wMap = new WeakMap();
let obj = {
    name:"Jaki"
}
//添加键值对
wMap.set(obj,true);
//判断某个键是否存在
console.log(wMap.has(obj));//true
//获取某个键的值
console.log(wMap.get(obj));//true
//删除一组键值对
wMap.delete(obj);
console.log(wMap.has(obj));//false
```

4.5 Proxy 代理

ES6 标准中定义了 Proxy 对象，从字面理解它是一种“代理”对象。Proxy 提供了一种途径，其允许开发者对已经存在的对象行为进行拦截与修改。

举一个小例子，在笔者还在上中学的时候，父母为笔者买了一台可以上网的电脑，但是他们像其他畏惧新事物（尤其是网络）的父母一样，担心笔者沉迷于网络而荒废学业，于是他们花大价钱在电脑中装了一个管理软件，只要笔者打开电脑，无论是看电影还是打游戏，他们都知道得一清二楚。其实这个管理软件就是计算机的一层代理，它不仅可以监控计算机的一举一动，还可以进行拦截与修改行为（比如父母觉得笔者看电影时间有些长，这些软件就能轻松地屏蔽视频播放功能）。

4.5.1 使用 Proxy 代理对对象的属性读写进行拦截

首先创建一个对象，点语法可以轻松地访问对象的属性，例如：

```
let teacher = {
    name:"Jaki",
    age:25,
    teaching:function(){
        console.log("teaching");
    }
}
console.log(teacher.name);//Jaki
```

下面我们使用 Proxy 对 teacher 对象属性的访问进行拦截：

```
let proxy_teacher = new Proxy(teacher,{
    set:(target,key,value,receiver)=>{
        console.log("添加属性:",key);
```

```
            target[key] = value;
        },
        get:(target,key,receiver)=>{
            console.log("获取属性:",key);
            return target[key];
        }
    });
    /*
    将打印
    获取属性: name
    Jaki
    添加属性: subject
    获取属性: subject
    JavaScript
    */
    console.log(proxy_teacher.name);
    proxy_teacher.subject = "JavaScript";
    console.log(proxy_teacher.subject);
```

Proxy 构造函数中需要传入两个参数，第 1 个参数为要代理的对象，第 2 个参数也是一个对象，其中需要定义要拦截或修改行为的方法，我们通常也会把它称为处理器对象。在对代理对象的某个属性赋值时，会触发处理器中的 set 方法，这个方法中前 3 个参数分别表示原对象、被赋值的属性名、所附的值。在读取代理对象的某个属性时会触发处理器中的 get 方法，这个方法中的前两个参数分别表示原对象和要访问的属性名。

需要注意，要使拦截方法起作用，必须使用 Proxy 代理对象，原对象的属性访问操作并不会触发代理对象的处理器方法。

抽 丝 剥 茧

Proxy 代理本身也是一种对象，对象可以通过原型链的方式来实现方法的继承，因此我们可以将 Proxy 代理作为对象的原型来实现未定义属性的拦截，示例如下：

```
var proxy_normal = new Proxy({}, {
  get: function(target, property) {
     console.log("warning:this property is undefined");
    return undefined;
  },
});
let obj = Object.create(proxy_normal);
obj.time;//warning:this property is undefined
```

4.5.2 Proxy 代理处理器支持的拦截操作

4.5.1 小节我们演示了对属性读和写的拦截操作，分别在处理器中定义 get 和 set 方法即可。处理器支持的拦截操作不只如此，下面的示例代码列出处理器对象中可以定义的拦截方法：

```
let teacher = {
    name:"Jaki",
    age:25,
    teaching:function(){
        console.log("teaching");
    }
}
let proxy_teacher = new Proxy(teacher,{
    //添加属性时会触发
    set:(target,key,value,receiver)=>{
        console.log("添加属性:",key);
        target[key] = value;
    },
    //获取属性时会触发
    get:(target,key,receiver)=>{
        console.log("获取属性:",key);
        return target[key];
    },
    //判断对象中是否包含某个属性时触发
    has:(target,key)=>{
        console.log("检查属性:",key);
        return key in target;
    },
    //输出对象属性时触发
    deleteProperty:(target,key)=>{
        console.log("删除属性:",key);
        delete target[key];
    },
    //拦截 getOwnPropertyNames()和 keys()方法
    ownKeys:(target)=>{
        console.log("获取所有自身属性");
        return Object.getOwnPropertyNames(target);
    },
    //拦截 defineProperty()和 definePropertys()方法
    defineProperty:(target,key,desc)=>{
        console.log("定义属性:",key);
        return Object.defineProperty(target,key,desc);
    },
    //拦截 preventExtensions()方法
```

```
        preventExtensions:(target)=>{
            console.log("抑制可扩展性");
            return Object.preventExtensions(target);
        },
        //拦截 getPrototypeOf()方法
        getPrototypeOf:(target)=>{
            console.log("获取对象原型");
            return Object.getPrototypeOf(target);
        },
        //拦截 isExtensible()方法
        isExtensible:(target)=>{
            console.log("获取对象可扩展性");
            return Object.isExtensible(target);
        },
        //拦截 setPrototypeOf()方法
        setPrototypeOf:(target,proto)=>{
            console.log("设置对象原型");
            return Object.setPrototypeOf(target,proto);
        },
        //拦截 call()和 apply 方法，用于函数对象
        apply:(target,object,arguments)=>{
            console.log("拦截 call、apply 方法");
            target.apply(object,arguments);
        },
        //拦截构造函数方法
        construct:(target,arguments)=>{
            return {};
        }
    });
    console.log("name" in proxy_teacher);//检查属性: name  true
    delete proxy_teacher.name //删除属性: name
    console.log(Object.getOwnPropertyNames(proxy_teacher)); //获取所有自身属性
[ 'name', 'age', 'teaching' ]
    Object.defineProperty(proxy_teacher,"name",{
        value:"Jaki",
        writable:true,
        configruable:true
    });//定义属性: name
    // Object.preventExtensions(proxy_teacher);//抑制可扩展性
    Object.getPrototypeOf(proxy_teacher);//获取对象原型
    Object.isExtensible(proxy_teacher);//获取对象可扩展性
    Object.setPrototypeOf(proxy_teacher,{sex:"男"});//设置对象原型
```

抽丝剥茧

in 是我们前边没有专门提到的一个运算符，使用它可以获取对象中是否包含某个属性。

关于使用 Proxy 来代理目标对象，有一点需要额外注意，Proxy 对象方法中的 this 和原对象方法中的 this 并不一致，它们指向的是两个不同的对象。例如：

```
var student = {
    name:"Lucy",
    print:function(){
        console.log(this===student);
    }
}
let proxy_student = new Proxy(student,{});
student.print();//true
proxy_student.print();//false
```

4.6 Promise 承诺对象

Promise 为 JavaScript 异步编程提供了一种实现思路。首先我们先来看一个生活中的小例子。在日常生活中，人们总是需要相互帮助的，比如你今天工作相当忙，由于前几天的放松导致领导发飙了，他要求你今天下班前必须把工作结果汇报给他，但是不久前你又与上大学的表弟约好了今天要把几本教材资料给他送去。领导要求的自然是要完成的，但是说好的事情也不能失约，于是你决定找一个朋友替你送书给表弟，你依然可以加班做自己的工作，朋友只需要在任务完成后把结果告诉你就好了。这就是一种承诺关系，类比于程序中是这样一种关系，即将任务交给某个对象完成，这个对象完成任务后再将结果反馈回来。当然，任务有可能执行失败，比如你的朋友没有找到表弟。无论任务执行成功与否，执行任务的对象都会将结果反馈回来，我们暂把这个对象叫做承诺对象，即 Promise 对象。

4.6.1 Promise 对象执行异步任务

通过前面的小例子你应该可以体会到，Promise 对象是极佳的异步编程方案。比如某个延时任务，我们可以把它交给 Promise 对象，这样既不会阻塞程序的正常执行，当延时任务执行完成后又可以第一时间做逻辑处理。示例代码如下：

```
//打印结果
/*
go...
任务直接完成
*/
let promise = new Promise(function(resolve,reject){
```

```
    setTimeout(()=>{
        console.log("任务直接完成");
    },3000);
});
console.log("go...");
```

使用 Promise 构造函数来进行 Promise 实例对象的创建，构造函数中需要传入一个函数作为参数，这个参数函数比较特殊，它有两个参数，分别用来触发 Promise 示例对象的任务执行完成消息和触发 Promise 示例对象的任务执行失败消息，后面我们会具体介绍，参数函数的函数体即是 Promise 实例对象需要执行的任务代码。从上面打印信息可以看出，Promise 实例对象一旦被创建，其中任务会自动开始执行，上面代码做了延时 3 秒的打印操作，并没有阻塞主程序代码的执行。

其实 Promise 实例对象有 3 种状态，分别是 pending（执行中）、fulfilled（执行完成）、rejected（执行失败）。这 3 种状态都是内部触发的，外部无法对其进行操作。上面的代码我们并没有对 Promise 任务执行完成情况进行监听，在实际开发中，一般不仅需要得到 Promise 任务执行的情况，甚至还需要获取到任务执行完成后的一些数据。Promise 实例对象的 then 方法用来监听执行情况并接收传递的数据。请看如下代码：

```
//打印结果
/*
go...
任务直接完成
Success
*/
let promise = new Promise(function(resolve,reject){
    setTimeout(()=>{
        console.log("任务直接完成");
        resolve("Success");
    },3000);
});
promise.then((success)=>{
    console.log(success);
},(error)=>{
    console.log(error);
});
```

then 方法中可以传入两个参数，第 1 个参数为当 Promise 任务执行完成时回调的方法，第 2 个参数为当 Promise 任务执行失败时回调的方法，第 2 个参数为非必需的。仅仅对 Promise 的执行情况进行监听并没有意义，我们还需要告诉 Promise 对象什么情况算任务执行完成，什么情况又算任务执行失败。在创建 Promise 对象时，我们提到过参数函数中的两个参数：resolve 和 reject。这两个参数实际上也是函数，我们只需要在函数体合适的地方对它们进行调用即可，例如上面的代码，调用了 resolve，这时 Promise 示例对象的状态便被置为已完成，之后会执行 then 方法监听的已完成情况下的回调函数。

除了 then 方法，Promise 实例对象中要定义一个 catch 方法，catch 方法和 then 方法的区别只在于 catch 方法只有一个参数，其是 Promise 执行失败后回调的函数，例如：

```
//打印结果
/*
go...
任务直接完成
error
*/
let promise = new Promise(function(resolve,reject){
    setTimeout(()=>{
        console.log("任务直接完成");
        reject("error");
    },3000);
});
promise.catch((error)=>{
    console.log(error);
});
```

4.6.2　Promise 任务链

在开发中还经常会遇到这样的场景，任务 B 的执行必须依赖于任务 A，即只有当任务 A 成功执行后才可以执行任务 B。使用 Promise 可以十分容易地实现这种逻辑，其实 Promise 实例对象的 then 方法中可以返回一个新的 Promise 来构成任务链，示例代码如下：

```
//打印结果
/*
go...
任务直接完成
success
任务 2 执行完成
success2
*/
let promise = new Promise(function(resolve,reject){
    setTimeout(()=>{
        console.log("任务直接完成");
        resolve("success");
    },3000);
});
promise.then((success)=>{
    console.log(success);
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            console.log("任务 2 执行完成");
            resolve("success2");
        },2000)
    });
```

```
},(error)=>{
    console.log(error);
}).then(success=>{
    console.log(success);
});
console.log("go...");
```

使用这种任务链的编程模式，我们可以很轻松地处理多任务并且有复杂依赖关系的场景。

4.6.3　Promise 对象组合

Promise 构造函数对象上还提供了两个非常有用的方法，它们用来对 Promise 任务进行组合，all 方法用来组合一组 Promise 实例对象，当组中所有的 Promise 任务都成功执行完成后，Promise 任务组才算成功执行，如果其中有一个任务执行失败，则任务组执行失败，例如：

```
let promise1 = new Promise((resolve,reject)=>{
    console.log("任务 1 执行成功");
    resolve();
});
let promise2 = new Promise((resolve,reject)=>{
    console.log("任务 2 执行成功");
    resolve();
});
let promise3 = new Promise((resolve,reject)=>{
    console.log("任务 3 执行成功");
    resolve();
});
let promiseGroup = Promise.all([promise1,promise2,promise3]);
/*
任务 1 执行成功
任务 2 执行成功
任务 3 执行成功
任务组执行成功
*/
promiseGroup.then(success=>{
    console.log("任务组执行成功");
},error=>{
    console.log("任务组执行失败");
});
```

如果我们将任务组中的一个单独任务修改为执行失败，则结果会完全不同，代码如下：

```
let promise1 = new Promise((resolve,reject)=>{
    console.log("任务 1 执行成功");
    resolve();
});
```

```
let promise2 = new Promise((resolve,reject)=>{
    console.log("任务 2 执行失败");
    reject();
});
let promise3 = new Promise((resolve,reject)=>{
    console.log("任务 3 执行成功");
    resolve();
});
let promiseGroup = Promise.all([promise1,promise2,promise3]);
/*
任务 1 执行成功
任务 2 执行失败
任务 3 执行成功
任务组执行失败
*/
promiseGroup.then(success=>{
    console.log("任务组执行成功");
},error=>{
    console.log("任务组执行失败");
});
```

与 all 方法对应的还有一个 race 方法，这个方法也是组合一组 Promise 实例对象，不同的是 race 方法组合的任务组中只要有一个任务执行完成，任务组就算执行完成，任务组的成功或失败只与第一个完成的任务有关，第一个完成任务的 Promise 对象状态如果为成功，则任务组的状态为成功，此对象的状态为失败，则任务组的状态为失败。例如：

```
let promise1 = new Promise((resolve,reject)=>{
    //加延时
    setTimeout(()=>{
        console.log("任务 1 执行成功");
        resolve();
    },1000);



});
let promise2 = new Promise((resolve,reject)=>{
    //加延时
    setTimeout(()=>{
        console.log("任务 2 执行成功");
        resolve();
    },1000);
});
let promise3 = new Promise((resolve,reject)=>{
    console.log("任务 3 执行失败");
    reject();
});
```

```
let promiseGroup = Promise.race([promise1,promise2,promise3]);
/*
任务 3 执行失败
任务组执行失败
任务 1 执行成功
任务 2 执行成功
*/
promiseGroup.then(success=>{
    console.log("任务组执行成功");
},error=>{
    console.log("任务组执行失败");
});
```

博 闻 强 识

Promise 构造函数对象的 all 方法返回的任务组，实际上其成功失败取决于其中每一个任务的执行。Promise 构造函数对象的 race 方法返回的任务组，其成功失败只与第一个执行完的任务有关，这样对比可以便于记忆。

4.7　Generator 生成器与 yield 语句

你已经掌握了 Promise 对象的应用，相信对编程中的异步处理任务有了更深入的理解。本节将向你介绍一种 ES6 中强大的任务执行控制方案：Generator 生成器对象。

从字面上理解，Generator 是一种生成器对象，这种理解并不确切但也不错误。一种更加面向应用的理解为 Generator 是一个状态机，其内部封装了多种状态，通过 yield 语句进行隔离。使用 Generator 函数，分步处理任务将更加容易。

4.7.1　Generator 函数应用

下面是一个简单的 Generator 函数例子：

```
function* generatorFunc(){
    console.log("任务一");
    yield;
    console.log("任务二");
    yield;
    console.log("任务三");
    return;
}
let g = generatorFunc();
let t1 = g.next();//任务一
let t2 = g.next();//任务二
```

```
let t3 = g.next();//任务三
let t4 = g.next();
console.log(t1);//{ value: undefined, done: false }
console.log(t2);//{ value: undefined, done: false }
console.log(t3);//{ value: undefined, done: true }
console.log(t4);//{ value: undefined, done: true }
```

我们先来看 Generator 函数的语法，其和普通函数十分相似，不同的是其在 function 关键字后追加一个星号表示它是一个 Generator 函数。Generator 函数内部可以编写一些逻辑代码，和普通函数不同，除了可以使用 return 关键字来返回外还可以使用 yeild 关键字来中断（普通函数中不可以使用 yield 关键字）。Generator 函数的调用和普通函数一样，但是有一点需要特别注意，调用 Generator 函数并非执行函数体的内容，而是返回一个迭代器对象，通过这个迭代器对象的 next 指针来分步执行 Generator 函数体中的任务。

以上面的代码为例，g 为调用 Generator 函数返回的迭代器对象，当迭代器对象第一次调用 next 方法时，其会执行 Generator 函数体的代码，直到遇到 yield 语句处中断。第 2 次再调用 next 方法时，Generator 函数体中的任务会从上次中断的地方开始继续执行，直到再次遇到 yield 中断或者 return 返回。当然，调用 next 函数时本身也会有一个返回值，这个返回值是一个对象，其中的 done 属性如果为 false，代表 Generator 函数体的任务还没有执行完成，可以继续调用 next 往后执行，如果 done 属性为 true，则表示 Generator 函数体的任务已经完全执行完成，之后再次调用 next 方法将不会有效果。从上面代码的打印结果你也可以看到，next 函数返回的对象中还有一个 value 属性，这个属性用于接收 Generator 函数体任务执行中使用 yield 或者 return 返回的数据，例如：

```
function* generatorFunc(){
    console.log("任务一");
    yield 1;
    console.log("任务二");
    yield 2;
    console.log("任务三");
    return 3;
}
let g = generatorFunc();
let t1 = g.next();//任务一
let t2 = g.next();//任务二
let t3 = g.next();//任务三
let t4 = g.next();
console.log(t1);//{ value: 1, done: false }
console.log(t2);//{ value: 2, done: false }
console.log(t3);//{ value: 3, done: true }
console.log(t4);//{ value: undefined, done: true }
```

再来看一种特殊的情况，如果我们要在一个 Generator 函数任务中执行另一个 Generator 函数任务，该如何来做呢，请看如下代码：

```
function* generatorSubFunc(){
    console.log("子任务一");
```

```
        yield;
        console.log("子任务二");
        return;
    }
    function* generatorFunc(){
        console.log("任务一");
        yield 1;
        console.log("任务二");
        let sub = generatorSubFunc();
        sub.next();
        sub.next();
        yield 2;
        console.log("任务三");
        return 3;
    }
    let g = generatorFunc();
    let t1 = g.next();//任务一
    let t2 = g.next();//任务二 子任务一 子任务二
    let t3 = g.next();//任务三
```

上面的示例代码提供了一种解决方案，但是代码结构并不优美，ES6 中提供了 yield*语句来直接在 Generator 函数体内部执行另一个 Generator 函数体任务，修改上面的代码如下：

```
    function* generatorSubFunc(){
        console.log("子任务一");
        yield;
        console.log("子任务二");
        return;
    }
    function* generatorFunc(){
        console.log("任务一");
        yield 1;
        console.log("任务二");
        yield* generatorSubFunc();
        console.log("任务三");
        return 3;
    }
    let g = generatorFunc();
    let t1 = g.next();//任务一
    let t2 = g.next();//任务二 子任务一 子任务二
    let t3 = g.next();//任务三
```

4.7.2 Generator 任务参数的传递

你知道通过 yield 或 return 语句可以得到 Generator 函数每一步任务的返回值，但是函数有三要

素：定义域、值域、表达式。对应于编程，其实就是参数、返回值、函数体。比如我们要实现这样的功能的 Generator 函数：第一个任务先传入两个参数，计算其和，然后输出结果；第二个任务是计算上一步结果的平方，再次输出；最后一个任务是传入一个参数，计算上一步结果与传入参数的差，再输出。其实我们在调用 next 方法的时候可以传递参数，配合 yield 语句来接收传递的参数，示例如下：

```
function* cal(){
    console.log("可以开始计算");
    let a = yield;
    console.log("接收参数 a",a);
    let b = yield;
    console.log("接收参数 b",b);
    let res = a+b;
    yield res;
    res = res*res;
    let d = yield res;
    console.log("接收参数 d",d);
    res = res - d;
    return res;
}
let c = cal();
c.next();
c.next(5);
console.log(c.next(3));//{ value: 8, done: false }
console.log(c.next());//{ value: 64, done: false }
console.log(c.next(10));//{ value: 54, done: true }
```

如上代码所示，当调用 next 函数的时候，我们可以传入一个参数，此参数会作为上一个任务 yield 表达式的值。切记，传入的参数会作为上一次任务的 yield 表达式的值，而不是本次任务 yield 表达式的值。也就是说，第一次调用 next 方法时实际上是不能传入参数的。

博 闻 强 识

当 Generator 函数作为对象的属性时，往往可以简写，如下面的代码所示：

```
var obj = {
    gFunc1:function* (){
        //...
    },
    * gFunc2(){
        //...
    }
}
```

obj 对象中的 gFunc1 与 gFunc2 都是 Generator 函数。

4.8　使用 class 定义类

JavaScript 是一种面向对象的语言，但是它并不是基于类的。通过前面的学习，你已经掌握了在 JavaScript 模拟类的方式，尽管我们可以通过对象冒充或者原型链的方式来实现类的功能，但这种定义类的方式与传统面向对象的语言有很大的不同，ES6 中新引入了 Class 关键字，使用它定义类将更加容易、更加规范。

4.8.1　使用 class 定义类

先来回顾一下，使用构造函数来定义类的方式：

```
function Teacher(name,age){
    this.name = name;
    this.age = age;
    this.teaching = ()=>{
        console.log(this.name,this.age);
    }
}
let teacher = new Teacher("Jaki",25);
teacher.teaching();//Jaki 25
```

ES6 的 class 关键字定义了一种类模板，使用 class 关键字来改写上面的 Teacher 类：

```
class Teacher{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
    teaching(){
        console.log(this.name,this.age);
    }
}
let teacher = new Teacher("Jaki",25);
teacher.teaching();//Jaki 25
```

下面我们来解释一下 class 关键字的用法。首先 class 关键字用于定义一个类，在代码块内需要提供一个 constructor 方法作为类的构造方法，也就是说，当使用 new 关键字加类名来创建实例化对象时，系统会调用 constructor 方法。除了 constructor 方法外，还可以在类中添加其他自定义方法，这些方法默认都会放入实例对象的原型对象中，换句话说，这些方法是所有实例对象所共享的。

抽 丝 剥 茧

不一定非要显式地实现 constructor 方法，如果不实现，那么系统会自动提供一个空的 constructor 方法作为构造函数。

4.8.2 class 类的继承

使用 class 关键字定义类的另一个方便之处在于继承对其来说十分容易。我们不需要再手动对原型链进行操作。例如：

```
class People{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
}
class Teacher extends People{
    constructor(name,age,subject){
        super(name,age);
        this.subject = subject;
    }
    teaching(){
        console.log(this.name,this.age,this.subject);
    }
}
let teacher = new Teacher("Jaki",25,"JavaScript");
teacher.teaching();//Jaki 25 JavaScript
```

class 的继承采用 extends 关键字，需要注意，如果你使用了继承，那么在 constructor 构造方法中必须先调用父类的构造方法，即使用 super 关键字来调用。super 关键字既可以当函数来使用，也可以当父类对象的原型对象来使用。如果在子类的构造方法中使用 super()，则它表示调用父类的构造方法，如果使用 super.method 的方式来调用父类的方法，则表示的是父类对象的原型对象，例如：

```
class People{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
    sayHi(){
        console.log("sayHi");
    }
}
class Teacher extends People{
```

```
    constructor(name,age,subject){
        super(name,age);
        this.subject = subject;
        super.sayHi();
    }
    teaching(){
        console.log(this.name,this.age,this.subject);
    }
}
let teacher = new Teacher("Jaki",25,"JavaScript");//sayHi
teacher.teaching();//Jaki 25 JavaScript
```

博 闻 强 识

使用 class 定义类可以实现的功能在 ES5 中基本也都可以实现，其实，ES6 中 class 这种语法只是引入了一种更加直观可读、更加规范的定义类的语法层面的实现，其实质依然是原型链。

4.9　模块引入

在 JavaScript 中，一直都缺少一种模块引入的语法，这使大项目的拆分变得十分困难。ES6 中引入了模块化的设计思想，通过 export 关键字可以将一个文件内的对象导出，在另一个文件中使用 import 可以实现其他文件中对象的导入。由于我们使用 Sublime Text 编辑器的 JavaScript 解释环境并不能支持 export 和 import 模块编程，因此本节我们只做语法的演示，后面的实战中会将这种语法语句使用起来。

4.9.1　export 关键字

ES6 中的模块化编程主要是通过 export 和 import 两个关键字实现。export 用于规定当前文件对外提供的接口，import 则是引入这些功能接口。

例如，我们创建一个命名为 module.js 的文件，如果我们要将其作为配置文件向外界提供一些用户配置信息的接口，就可以这么做：

```
export var userName = "Jaki";
export var password = "123456";
```

上面的示例代码在变量的声明前添加了 export 关键字，其向外部输出了 userName 和 password 两个变量。当然你也可以选择一次性导出多个变量，例如：

```
var scrool = "No.1";
var theClass = "No.2";
export {scrool,theClass};
```

除了可以将变量作为导出的对象，也可以导出函数或者类，例如：

```
export function(){
    console.log("Hello World");
}
export class Teacher{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
}
```

默认情况下，导出的对象在外界的名称就是对象原始的名称，你也可以对导出对象的名称使用 as 进行自定义，例如：

```
var jaki = "Jaki";
var lucy = "Lucy";
export {jaki as people1,lucy as people2};
```

对于上面的示例代码，外界在使用 jaki 和 lucy 变量时，需要使用 people1 与 people2。

4.9.2 import 关键字

import 关键字和 export 关键字对应，用于引入 export 关键字所导出的对象。我们再创建一个名为 main.js 的文件，比如要导入 module.js 中的 userName 和 password 变量，就可以这么做：

```
import {userName,password} from "./module.js";
console.log(userName,password);
```

import 后面的大括号中需要指定要引入模块中导出的对象，from 关键字后面需要指定模块文件的路径。同样，在引入文件时，也可以为引入的对象起一个新的名字，例如：

```
import {userName as name,password as word} from "./module.js";
console.log(name,word);
```

improt 还提供了一种语法，其可以将外部模块中所有导出的对象一起引入进来并且绑定到指定的对象上，例如：

```
import * as module from "module.js";
console.log(module.userName,module.password);
```

4.9.3 默认导出与导入

很多时候，一个外部的模块往往只需要向外提供唯一的一个接口。例如通常将一个独立的类写成一个单独的文件，那么只暴露这个类本身作为接口就可以了。因此，ES6 的导出与导入还提供了 default 默认项设置。例如，在 module.js 文件中编写如下代码：

```
export default class People(){
    constructor(){
    }
}
```

上面代码的意思是将 People 类作为 module.js 文件的默认导出对象。在 main.js 中可以使用如下代码进行导入：

```
import myModule from "module.js";
```

上面的示例代码有两点需要注意，首先 myModule 是自定义的名称，代表 module.js 文件中默认导出的对象，即 People 类。还有一点需要特别注意，这种当时引入的对象是不需要写在大括号内的。默认对象和非默认对象也是可以同时引入的，它们并不会相互影响，例如：

```
import myModule,{userName} from "module.js";
```

第 5 章

React Native 开发环境的搭建

React Native 是由 Facebook 公司开源的一套跨平台的移动端开发框架。React Native 能够使你用 JavaScript 开发出一流体验的移动原生应用。虽然目前市场上纯 React Native 的应用数量还十分有限，但是一套代码跨平台使用的特点以及 Facebook 将持续不断地投入 React Native 建设都将使其前景十分美好。

React Native 开发的原生项目可以同时支持 iOS 与 Android 双平台。由于 iOS 开发工具 Xcode 软件和模拟器都只能运行在 macOS 系统上，因此本书也采用 macOS 系统来做演示。

5.1 iOS 开发环境的搭建

Xcode 开发工具是最全面、最强大、集成工具最完善的一款 iOS 软件开发工具。除了可以开发 iOS 应用外，还可以开发 macOS、watchOS、tvOS 平台的软件。Xcode 工具是 Apple 自家开发的工具，因此其下载安装也十分方便，直接使用 AppStore 软件即可完成下载与安装。

5.1.1 申请 AppleID 账号

从 AppStore 下载软件都需要使用 AppleID 先登录。本小节简单介绍如何申请 AppleID 账号，如果你已经拥有账号，完全可以跳过此章节向后学习。

首先登录 https://appleid.apple.com 网站来到 AppleID 管理页面，单击右上方的“创建您的 Apple ID”选项，如图 5-1 所示。

之后会跳转到 AppleID 创建界面，根据提示填写完整信息后完成注册即可。需要注意，填写的邮箱地址要务必真实有效，需要邮箱验证才能注册成功。还有一点，安全提示问题及答案要也牢记，万一有一天忘记了自己的 AppleID 密码，这些可以帮助你将其找回，如图 5-2 所示。



图 5-1　创建 AppleID 账号

只需一个 Apple ID，您即可访问 Apple 所有内容。
已有一个 Apple ID？在此查找›
name@example.com
密码
确认密码
姓氏
名字
出生日期
安全提示问题 1
答案
安全提示问题 2
答案

图 5-2　填写申请 AppleID 的基本信息

5.1.2　安装 Xcode 开发工具

打开 AppStore 软件，在其搜索栏中输入 Xcode 关键字按回车键进行搜索，如图 5-3 所示。

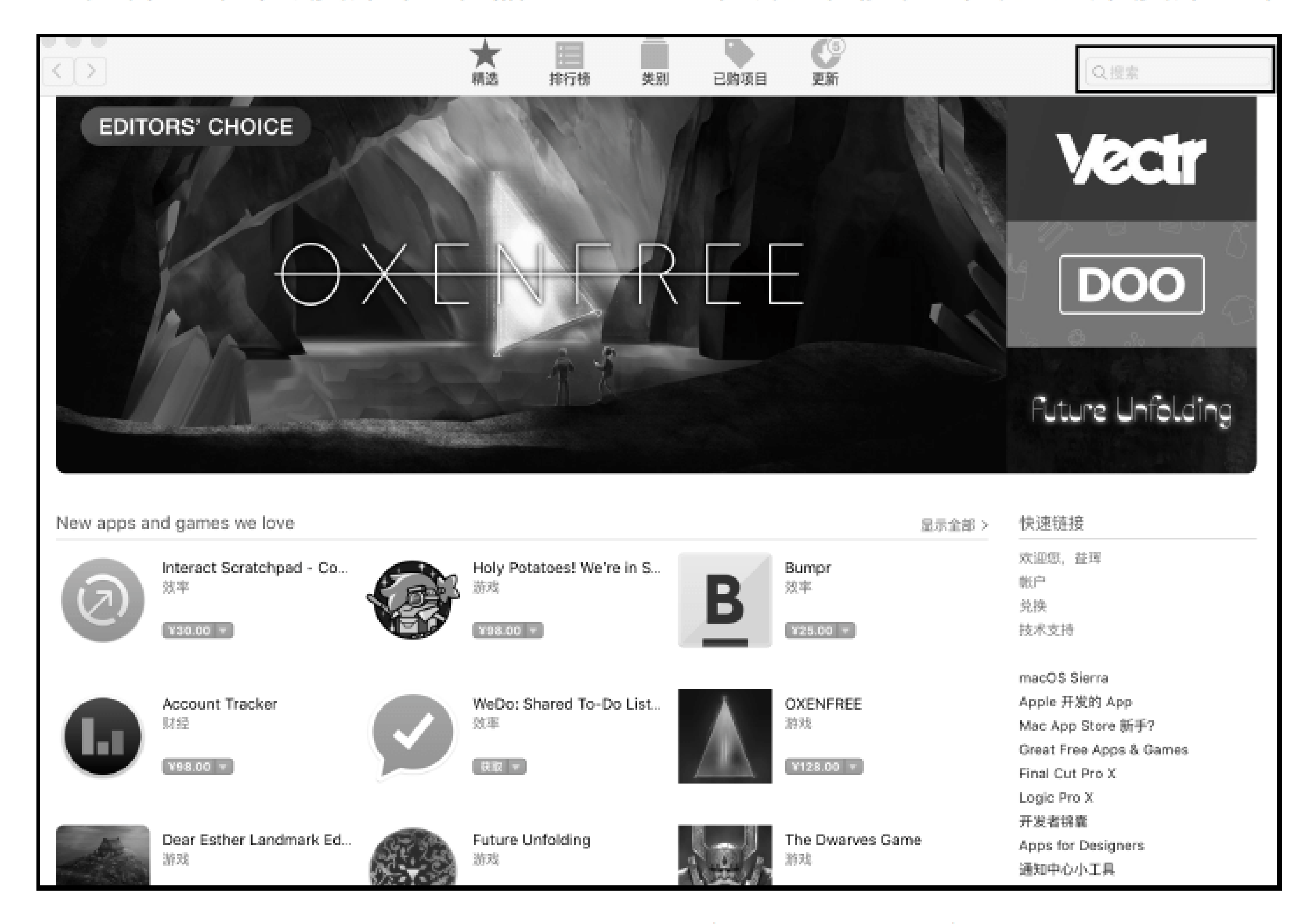


图 5-3　在 AppStore 中进行软件搜索

搜索结果中的第一个软件就是 Xcode 开发工具，单击获取即可进行下载安装，其间可能会弹出需要输入 AppleID 账号密码的菜单，正确填写即可。注意，如果你已经安装过 Xcode 工具，则这里显示的是打开而不是获取，如图 5-4 所示。

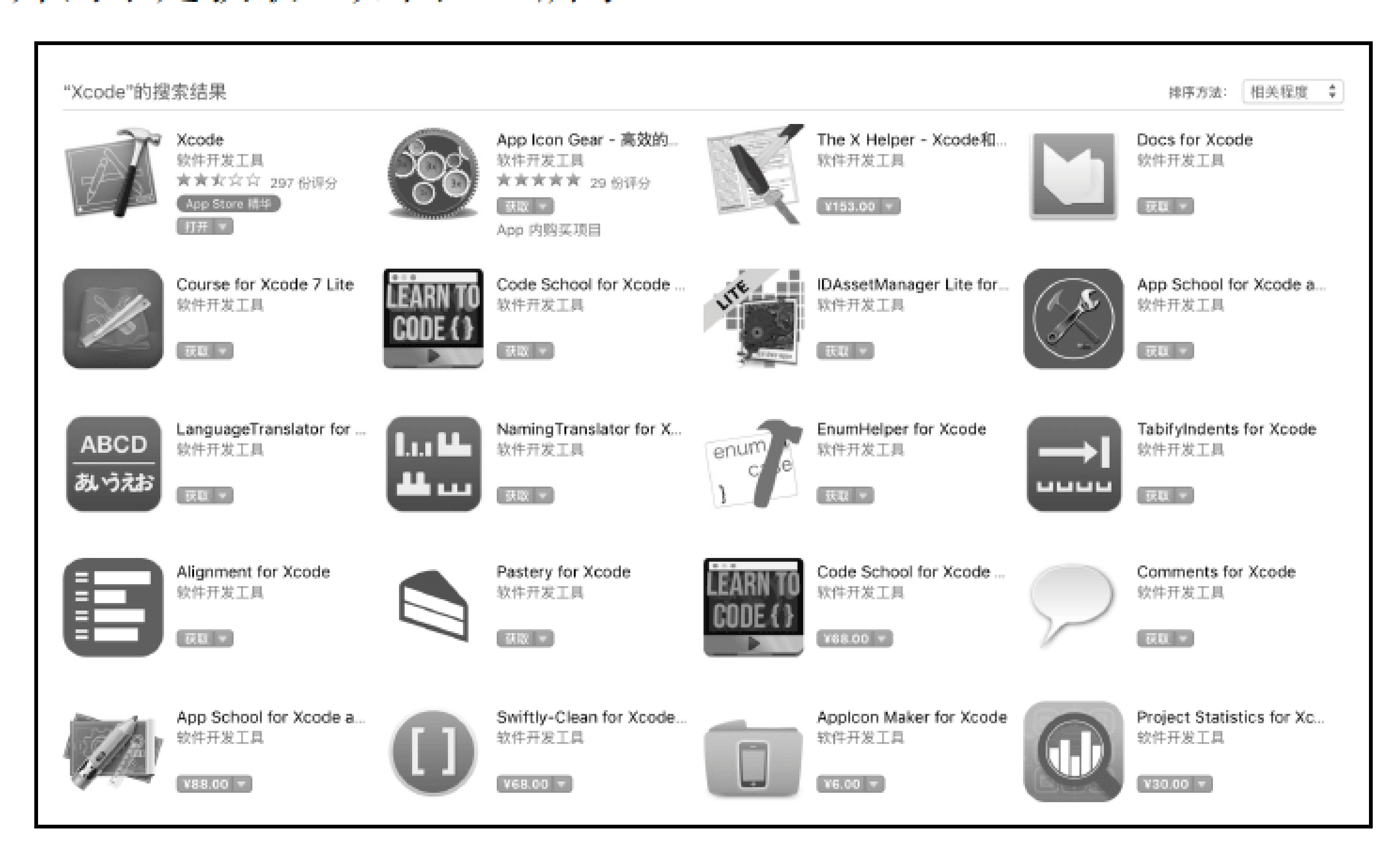


图 5-4　安装 Xcode 开发工具

5.2　Android 开发环境的搭建

以前国内的开发者若想直接下载 Android Studio 开发工具并不容易，现在 Google 专门为中国开发者提供了资源网站，无论是下载工具还是获取最新的 Android 开发资讯，都比以前容易了许多。

5.2.1　下载 Android Studio 开发工具

首先登录 Google 中国开发者网站，网址是 https://developers.google.cn/。单击其中的 Android 按钮，如图 5-5 所示。



图 5-5　Google 中国开发者网站

在弹出的窗口中选择“下载 Android Studio 和 SDK 工具选项”，如图 5-6 所示。

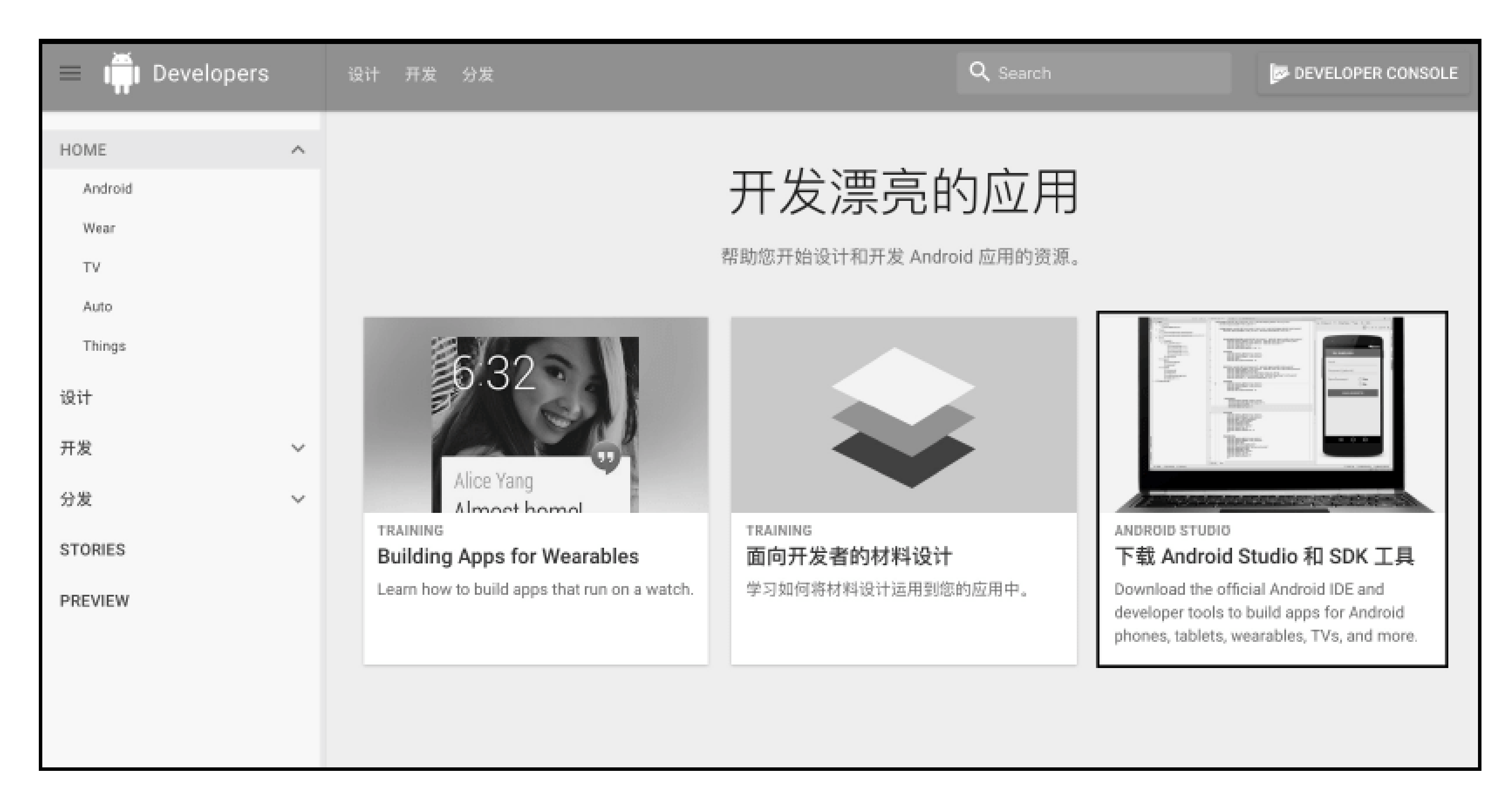


图 5-6　Android 开发者页面

在弹出的窗口中选择“下载 ANDROID STUDIO”，完成下载即可，如图 5-7 所示。



图 5-7　下载 Android Studio 开发工具

5.2.2　安装相关 SDK 和模拟器

下载安装完 Android Studio 后，你还需要安装相关 SDK 和模拟器，打开 Android Studio 开发工具，在欢迎界面中单击 Configure，选择其中的 SDK Manager，如图 5-8 所示。

在弹出窗口中选择 Android SDK，在 SDK Platforms 选项卡中勾选一个 Android SDK 版本进行安装即可，如图 5-9 所示（图中安装了 Android 6.0 与 Android 4.4 两个版本）。

然后在 SDK Tools 选项卡下选择 Android SDK Tools 23.0.1（必须是这个版本）和 Android Support Repository 进行安装。注意，要选择 SDK Tools 的版本，需要勾选 Show Package Details 选项。

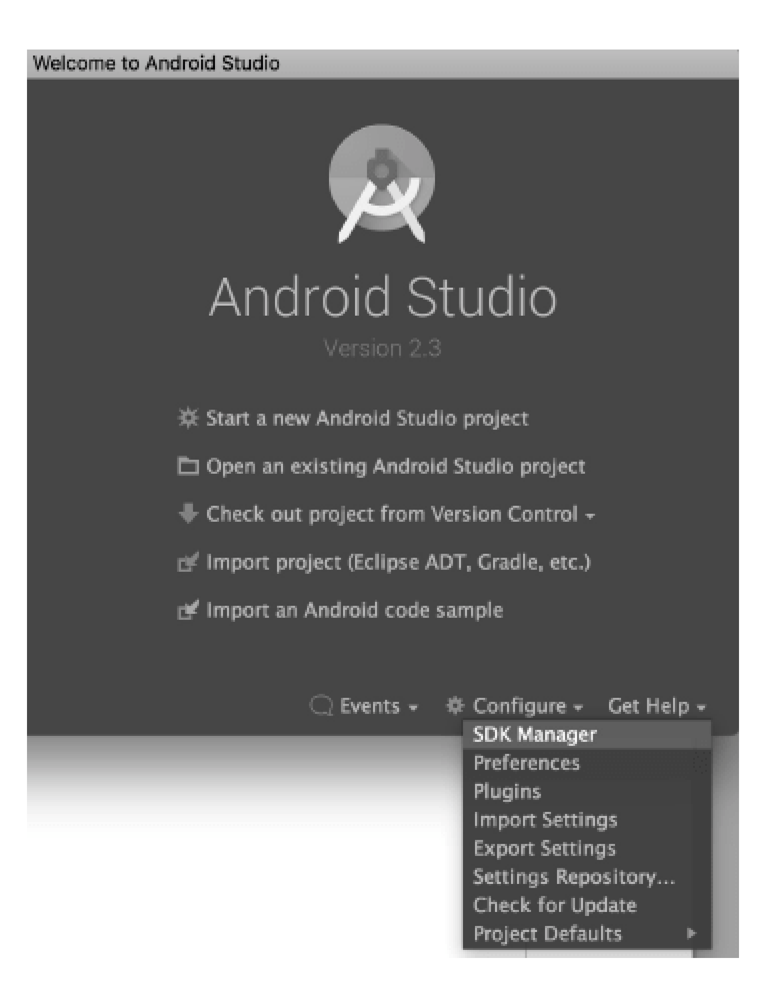


图 5-8　进行 SDK 下载与安装

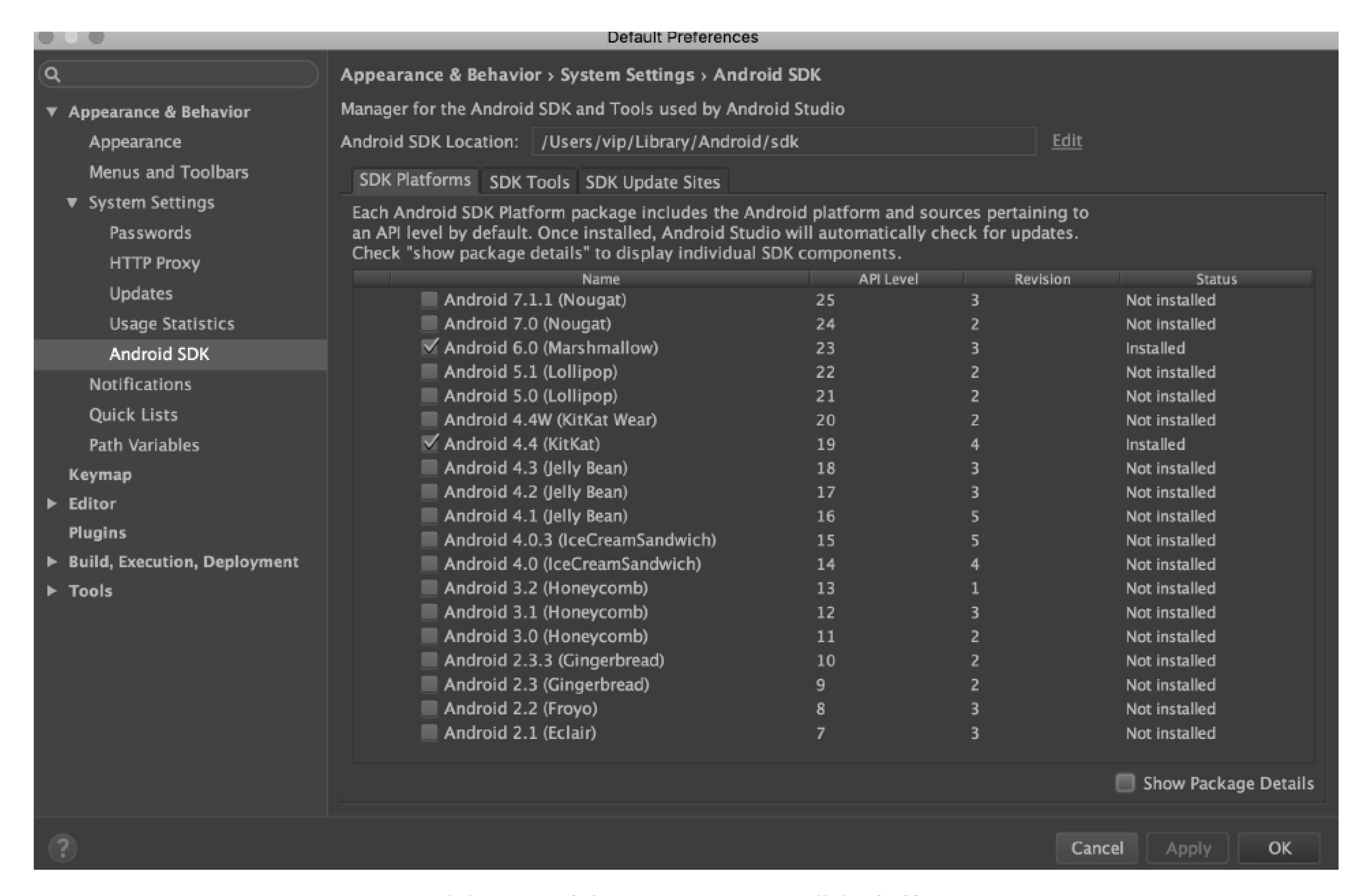


图 5-9　选择 Android SDK 进行安装

Android 模拟器的安装也十分简单，先随便新建一个 Android 工程，之后在菜单栏中找到 Tools→Android→AVD Manager 选项，如图 5-10 所示。

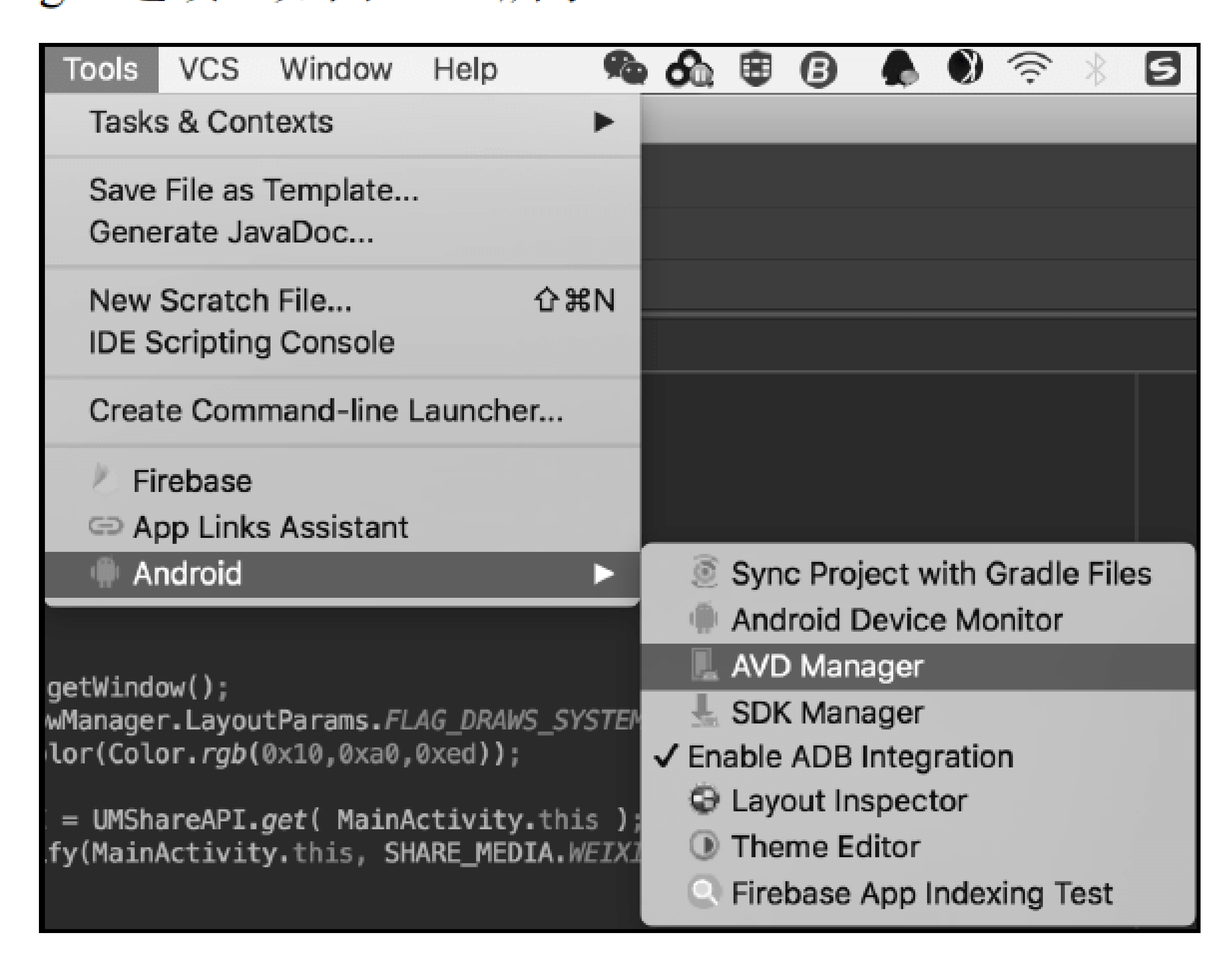


图 5-10　打开 Android 模拟器管理器

弹出的列表中为当前已有的模拟器，如果没有，单击 Create Virtual Device...选项创建一个，如图 5-11 所示。

在弹出的窗口中选择设备类型，如图 5-12 所示。

选择完设备类型后，单击 Next 按钮进入下一步，还需要为模拟器选择一个系统（如果还没有模拟器系统，单击 Download 下载一个即可），如图 5-13 所示。

选择完系统后单击 Next 按钮进入下一步，还需要我们进行模拟器的相关配置，一般不需要修改，直接单击 Finish 按钮完成模拟器的创建即可。

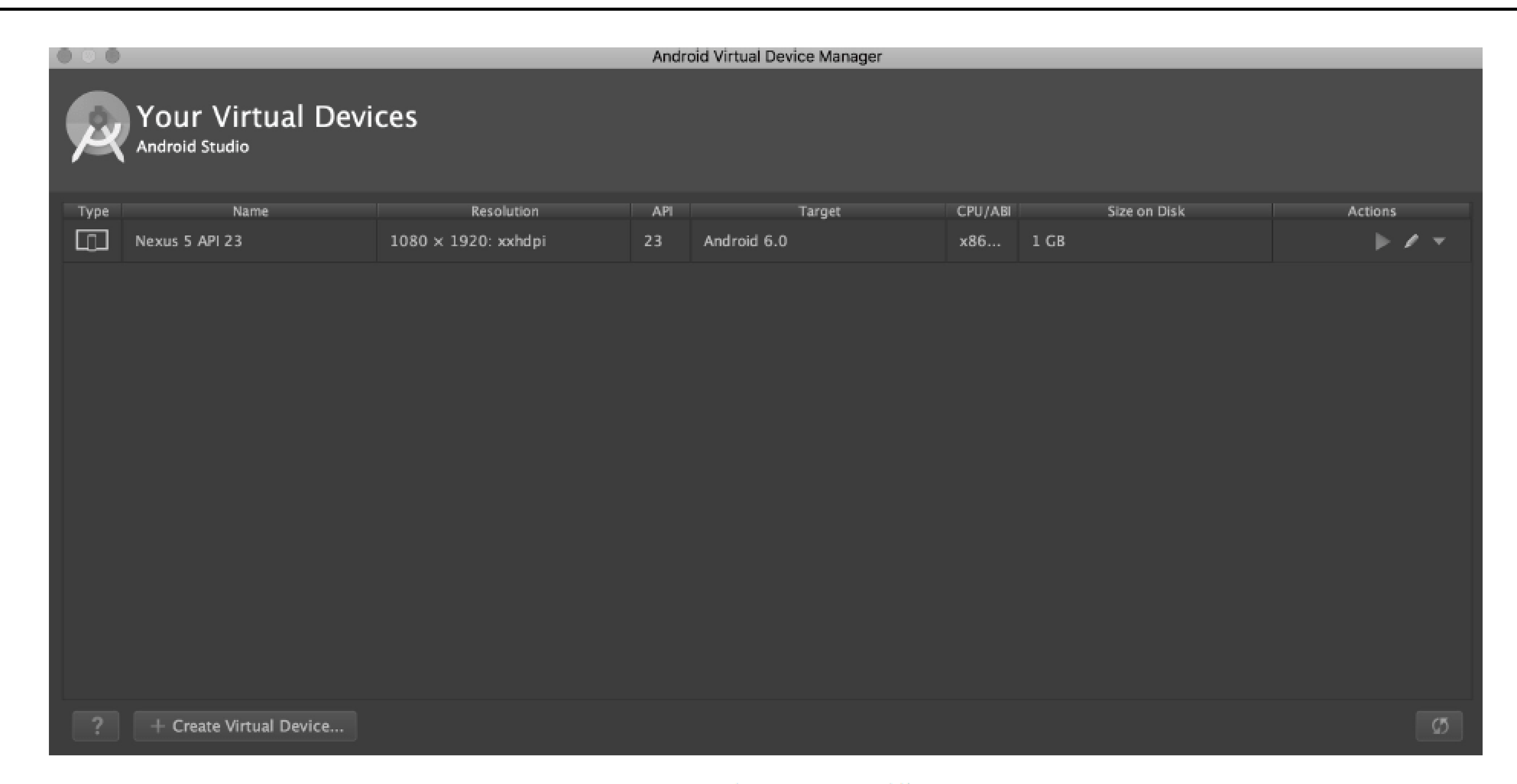


图 5-11　创建 Android 模拟器

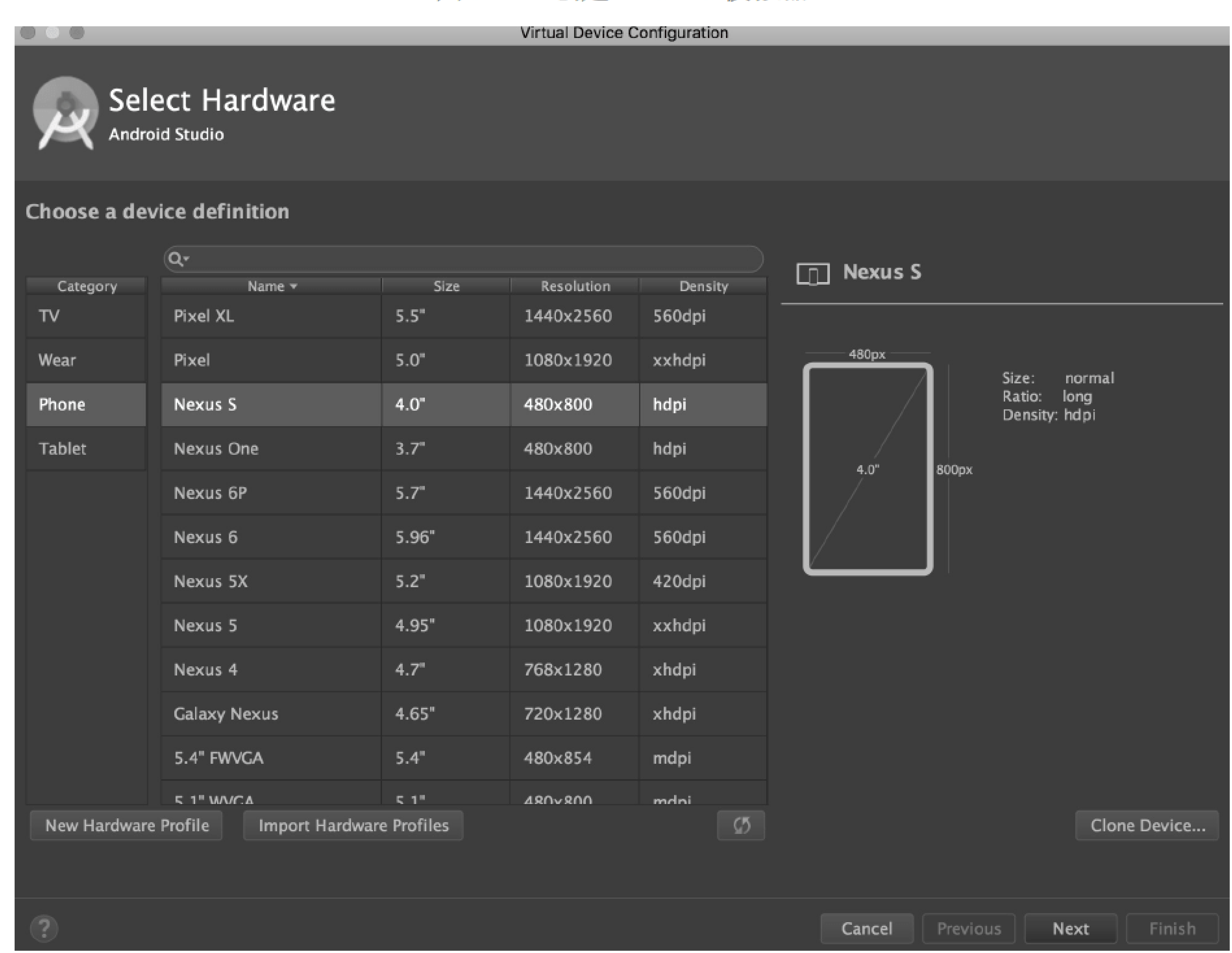


图 5-12　选择 Android 模拟器设备类型

现在 Android 开发环境的配置还差最后一步，需要将 ANDROID_HOME 环境变量正确地指向 Android SDK 路径，在终端输入如下命令：

```
vi ~/.bash_profile
```

在打开的文件中输入下面两行内容：

```
export PATH=$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools
export ANDROID_HOME=~/Library/Android/sdk
```

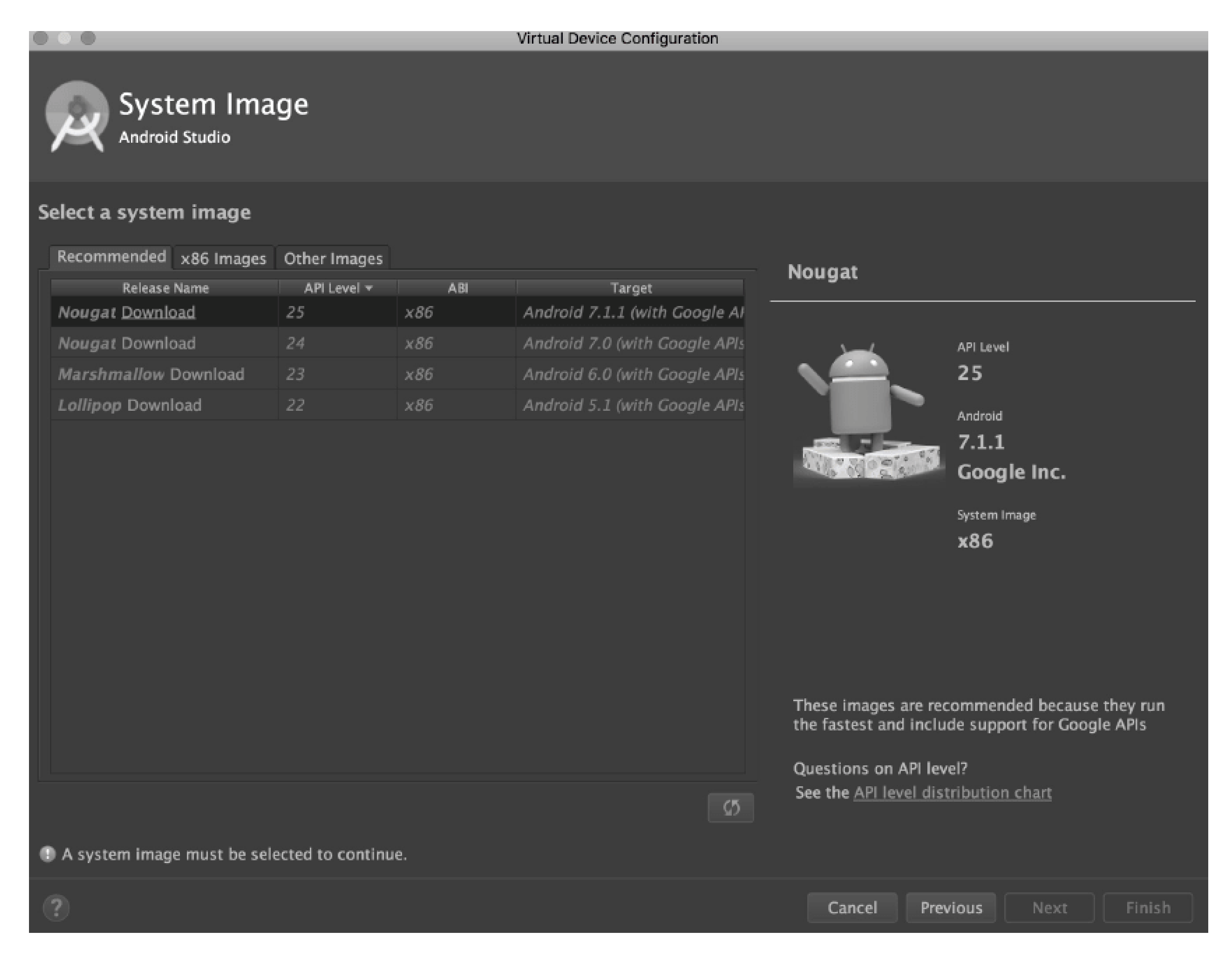


图 5-13　选择模拟器系统

之后进行保存即可。

抽 丝 剥 茧
bash_profile 文件可能会不存在，vi 命令会新建一个文件，写入内容后，按 Esc 键，之后使用 “shift+:” 进入命令模式，输入 wq 后回车保存即可。

5.3　React Native 开发环境配置

经过前面的步骤，终于将 iOS 开发环境和 Android 开发环境搭建完成，其实你现在已经可以独立地在各个平台上开发各自的软件，但这并不是我们本书的目的，下面就来安装 React Native 相关工具。

5.3.1　安装 React Native 构建环境

构建 React Native 工程，首先需要安装 Node.js 环境，其实前面我们在使用 Sublime Text 工具运行 JavaScript 代码时，就已经使用到了 Node.js，如果此时你还没有安装 Node.js，那么你需要回到第 1 章，先将准备工作完成。

我们还需要安装 React Native 的命令行工具：react-native-cli。在终端输入如下命令：

```
npm install -g react-native-cli
```

安装完成后可以使用如下命令来查看 React Native 的版本，如果查看到版本，就说明安装成功。

```
react-native -v
```

5.3.2　运行你的第一个 React Native 应用

下面让我们一起来搭建一个 React Native 版本的 HelloWorld 应用。在终端执行如下命令进行 React Native 工程的初始化：

```
react-native init HelloWorld
```

这个命令 init 后面的参数用来设置工程名称，这里取名为 HelloWorld。这个命令是一个集成命令，其会完成项目的创建以及依赖包的下载安装，因此会需要一定时间，你一定要有足够的耐心。安装完成后，进入 React Native 工程目录，使用 react-native run-ios 或者 react-native run-Android 即可在模拟器上运行 iOS 或者 Android 项目。如果你的 iOS 或者 Android 模拟器显示如图 5-14 或图 5-15 所示的界面，那么恭喜你，你的第 1 个 React Native 应用已经成功运行起来了。

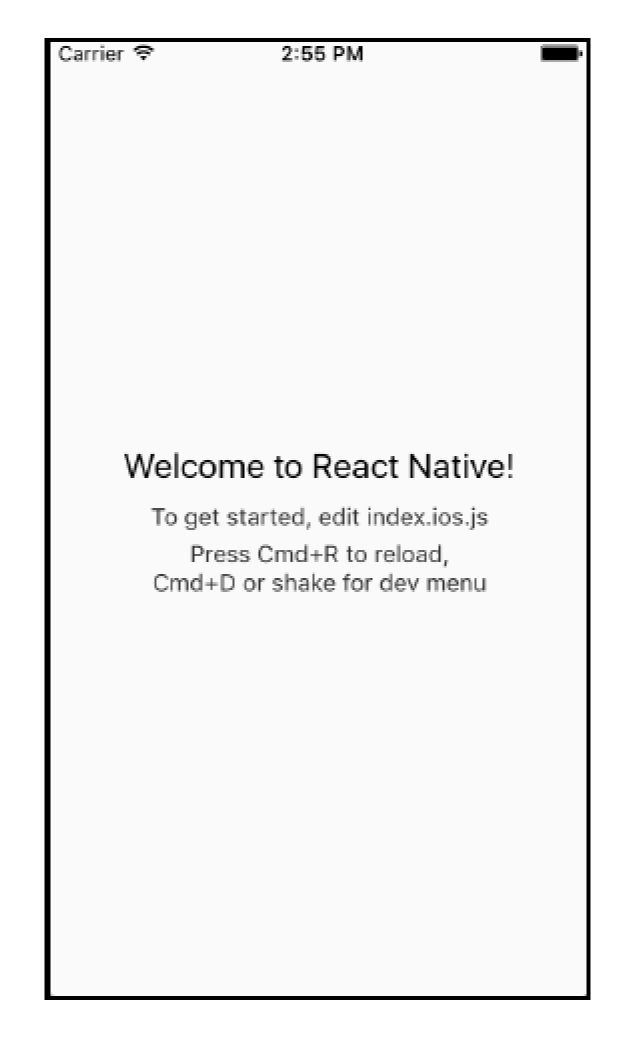


图 5-14　iOS 模拟器

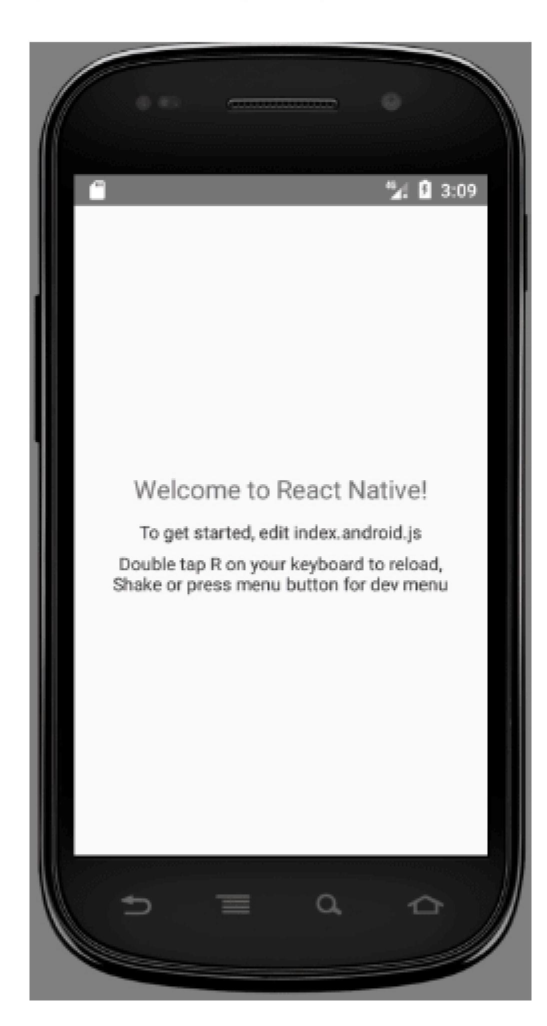


图 5-15　安卓模拟器

在运行 iOS 工程时会启动 Xcode 默认的模拟器，使用如下命令可以对要启动的模拟器进行选择：

```
react-native run-ios --simulator "iPhone SE"
```

上面命令的“iPhone SE”为 iOS 模拟器的名称，在终端使用如下命令可以查看所有支持的模拟器名列表：

```
xcrun simctl list devices
```

在运行 Android 工程时，则会自动运行在所开启的模拟器上，如果有使用数据线连接真机，则会运行在真机上。

打开 HelloWorld 工程目录，项目结构如图 5-16 所示。

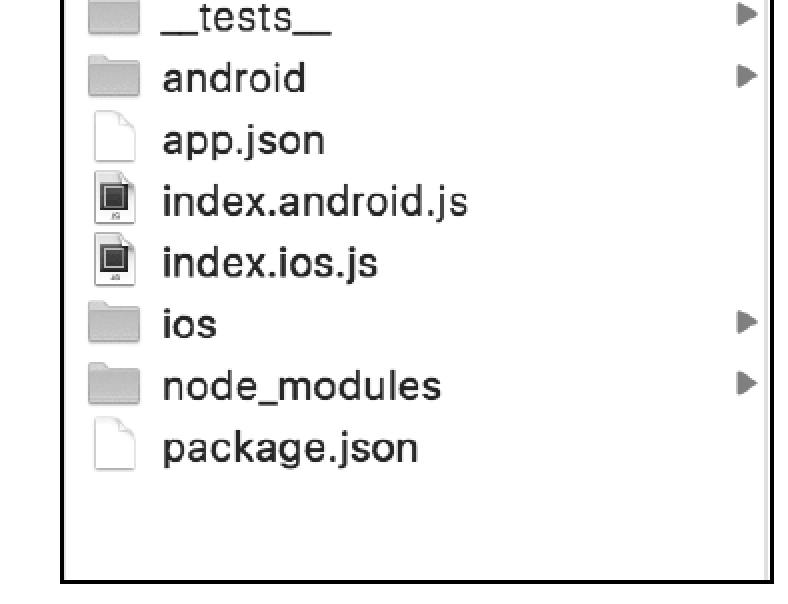


图 5-16　React Native 工程结构

在 React Native 工程结构中，node_modules 目录中为所有的依赖包，我们不需要关心。Android 文件夹是 Android 的项目目录，ios 文件夹是 iOS 项目目录。index.ios.js 文件和 index.Android.js 文件分别是 iOS 项目的入口文件与 Android 项目的入口文件。打开 index.ios.js 文件和 index.Android.js 文件，将其中的代码修改成如下所示：

```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
  Text,
  View
} from 'react-native';
export default class HelloWorld extends Component {
  render() {
    return (
      <Text style={{
        flex:1,
        top:100,
        left:100,
        fontSize:30
      }}>HelloWorld</Text>
    );
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

刷新 iOS 模拟器和 Android 模拟器，效果分别如图 5-17 与图 5-18 所示，你的 HelloWorld 项目完成了！

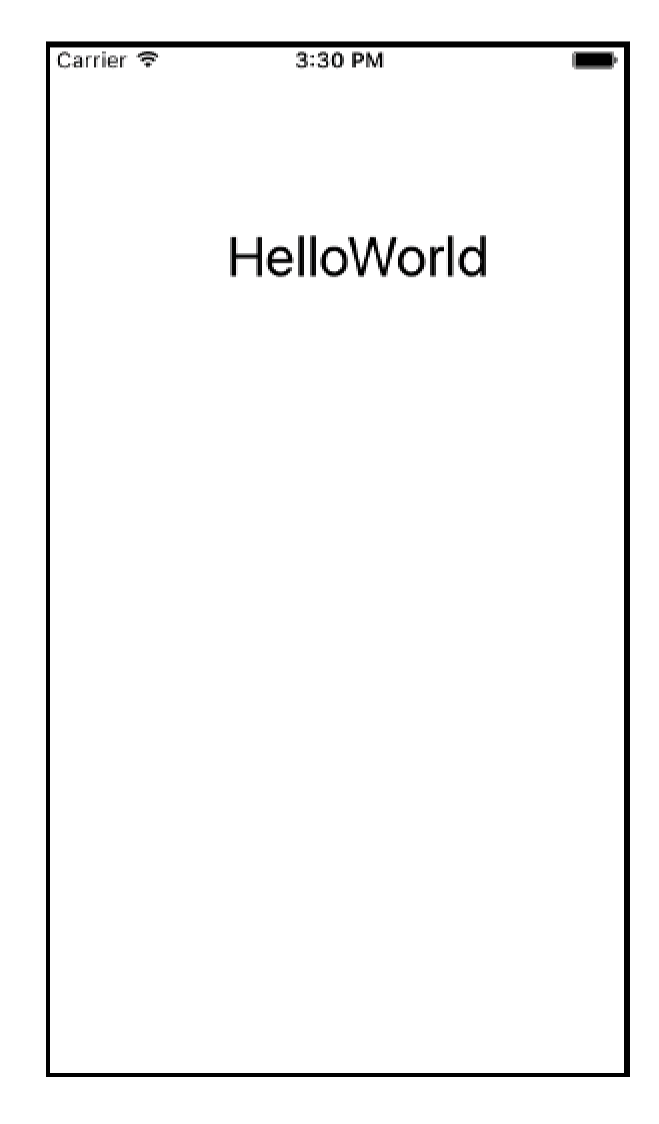


图 5-17　iOS 模拟器

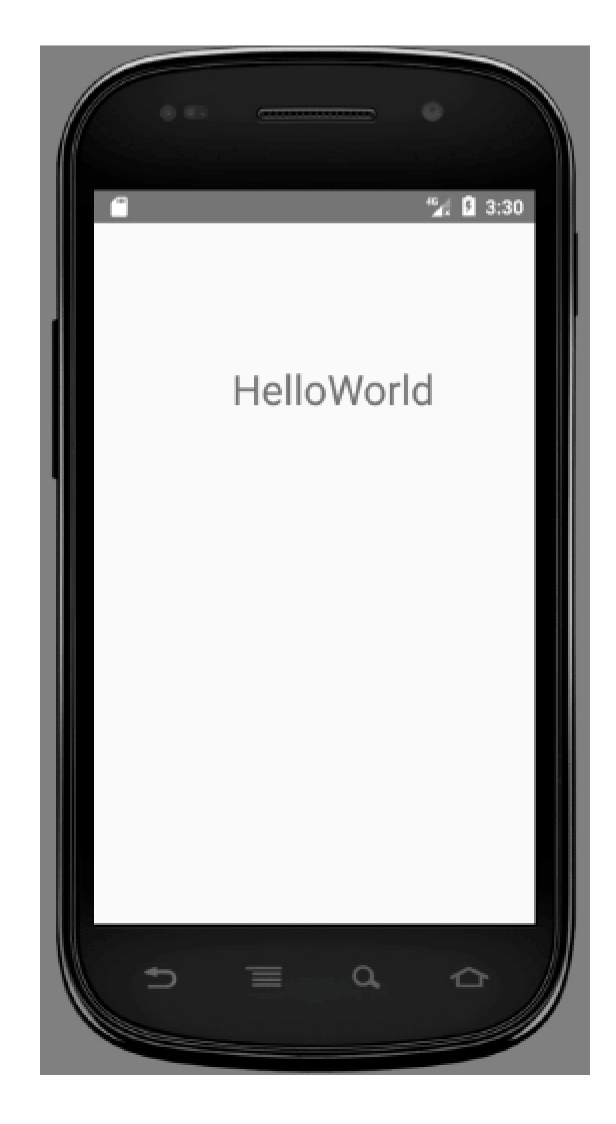


图 5-18　Android 模拟器

抽 丝 剥 茧
在 iOS 模拟器中使用 command+R 来进行刷新，在 Android 模拟器中双击 R 来进行刷新。

至此，搭建一个完整的 React Native 工程算是告一段落了。关于 index.ios.js 与 index.Android.js 文件中代码的意义，我们先不做过多解释，后面会具体讲解。由于 React Native 工程运行在 iOS 模拟器与 Android 模拟器上的效果类似，在后面的学习过程中，我们就只给出 iOS 模拟器的效果图作为展示。每次新建 React Native 项目时，都需要经过本章所讲解的初始化步骤，初始化过程是一个非常耗时的操作，因此后面在学习 React Native 开发基础时，我们会一直使用这个 HelloWorld 工程作为模板，创建不同的 JavaScript 文件来渲染界面。最后你可以随意修改一下 index 文件中的代码，观察效果。

第 6 章

React Native 独立组件基础篇

本章我们将介绍 React Native 中定义的独立开发组件。开发一款完整的应用其实就是使用各种逻辑将各个独立的组件进行连接与组合，如果你在本章的学习中熟练掌握了这些独立组件的使用，那么在后面的学习中将如鱼得水，游刃有余。

6.1 Text 文本组件的应用

应用程序中随处可见各种各样的文字，Text 文本组件用于在视图上显示文字，其通用的 API 可以使你轻松地在 iOS 平台与 Android 平台上渲染体验一流的文字效果。

6.1.1 文字风格设置

首先打开第 5 章创建的 HelloWorld 工程，在其中创建一个新的文件夹，命名为“Demo”，之后我们的示例文件都将放在这个文件夹中。在 Demo 文件夹中新建一个文件，命名为 TextDemo.js，作为本节示例文件。在 React Native 中大部分独立组件都有 style 这样一个属性，用来设置组件的风格。一般组件的风格设置包括两部分：公共部分与定制部分。公共部分的风格属性是继承于最基本的视图组件 View 而来，包括设置组件的位置、尺寸、背景色等，这不是我们本节的重点，后面我们会做具体的介绍。定制部分用来设置组件某些独特的属性，例如 Text 组件可以设置字体、文字颜色、字体对齐方式等。

在 TextDemo.js 文件中编写如下代码：

```
/*
文本控件 Text
*/
```

```
import React,{Component} from 'react';
import {Text} from 'react-native';
export default class TextDemo extends Component {
    render(){
        return (
            //Text 控件
            <Text style={[style,base]}>Hello Friend!</Text>
        );
    }
};
//公共样式
var base = {
   //设置控件距离上方 100 的单位
    top:100
};
//Text 组件特有的样式
var style = {
   //设置文字颜色
    color:'red',
    //设置字体
    fontFamily:'Cochin',
    //设置字号
    fontSize:24,
    //设置字体风格
    fontStyle:'italic',
    //设置字体粗细
    fontWeight:'bold',
    //设置行高
    lineHeight:50,
    //设置文字对齐模式
    textAlign:'center',
    //设置文字修饰风格
    textDecorationLine:'underline',
    //设置文字阴影颜色
    textShadowColor:'green',
    //设置阴影偏移
    textShadowOffset:{
        width:5,
        height:5
    }
};
//iOS 平台特有的样式
var iosStyle = {
    //设置文字变体
    fontVariant:['small-caps'],
```

```
        //设置字符间距
        letterSpacing:10,
        //设置修饰线颜色
        textDecorationColor:'blue',
        //设置修饰风格
        textDecorationStyle:'double',
        //设置书写方向
        writingDirection:'rtl'
    }
    //Android 平台特有的样式
    var androidStyle = {
       //设置是否有内间距
        includeFontPadding:false,
        //设置水平方向的对齐模式
        textAlignVertical:'center'
    }
```

将 index.ios.js 与 index.Android.js 文件修改如下：

```
    import React, { Component } from 'react';
    import {
      AppRegistry
    } from 'react-native';
    import TextDemo from './Demo/TextDemo';
    export default class HelloWorld extends Component {
      render() {
        return (<TextDemo />);
      }
    }
    AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

在终端运行工程，可以看到模拟器效果如图 6-1 所示。

由于 iOS 与 Android 平台的差异性，Text 组件风格设置中也有一些属性是只针对 iOS 平台或者只针对 Android 平台的，如上面代码中的 iosStyle 和 AndroidStyle，将这两个风格对象添加到 Text 组件中，代码如下：

```
    <Text style={[style,base,iosStyle,androidStyle]}>
Hello Friend!</Text>
```

刷新两平台模拟器，效果分别如图 6-2 与图 6-3 所示。

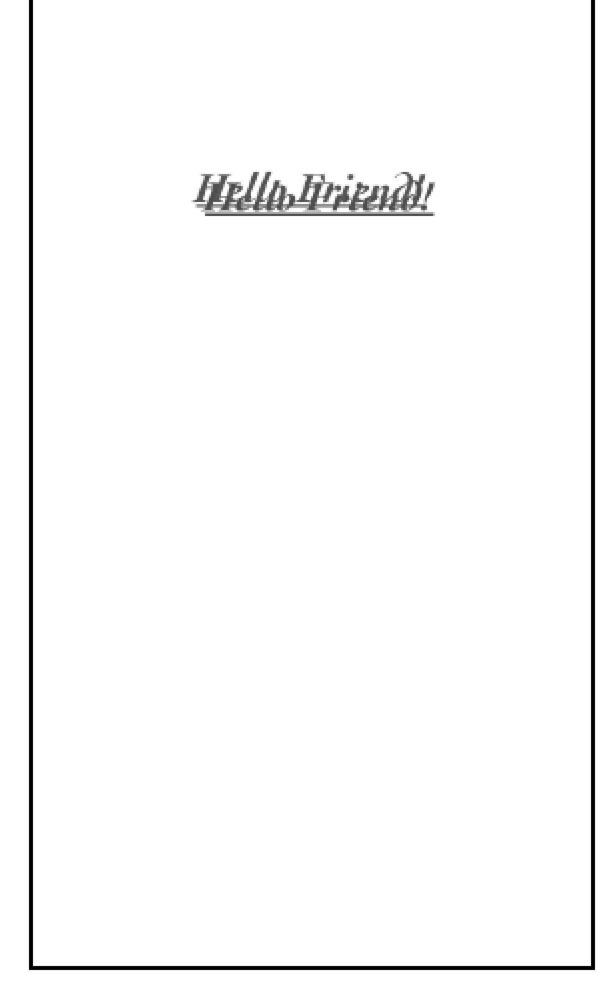


图 6-1　Text 组件效果

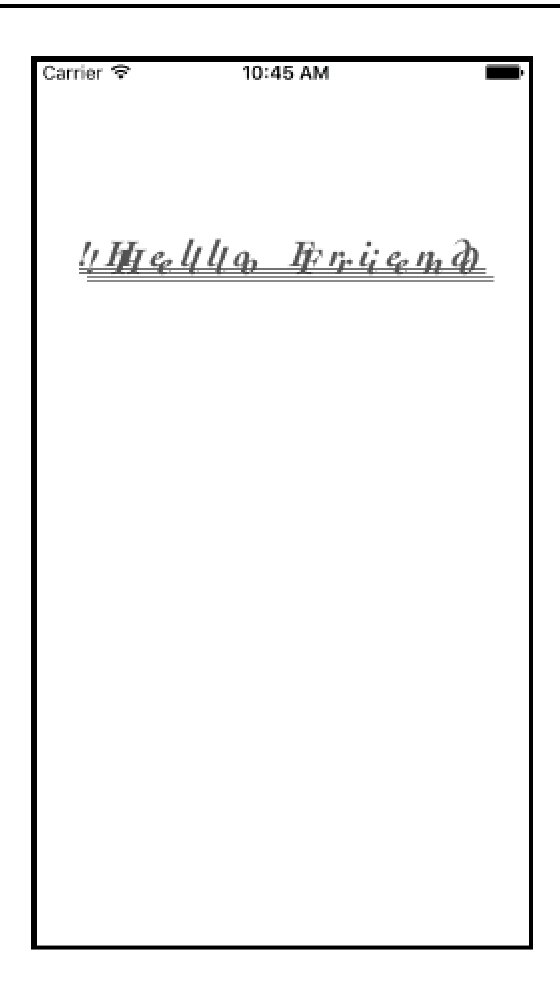


图 6-2　iOS 平台效果



图 6-3　Android 平台效果

表 6-1 列出了 Text 组件 style 属性可以进行设置的文本风格。

表 6-1　Text 组件 style 属性

属性名	解释	平台	值类型或可选值
color	文字颜色	通用	特定格式的 color 值
fontFamily	字体名称	通用	字体名称字符串
fontStyle	字体风格	通用	可选字符串： • normal：正常 • italic：斜体
fontWeight	字体粗细	通用	可选字符串： • normal：正常 • bold：粗体 也可以手动设置 100 到 900 间的一个整百数值字符串
lineHeight	文本行高	通用	数值
textAlign	文本对齐方式	通用	可选字符串： • auto：自动 • left：左对齐 • right：右对齐 • center：居中 • justify：正向（仅 iOS 可用）
textDecorationLine	文字修饰	通用	可选字符串： • none：无修饰 • underline：下画线 • line-through：删除线 • underline line-through：下画线与删除线

（续表）

属性名	解释	平台	值类型或可选值
textShadowColor	文字阴影颜色	通用	特定格式的 color 值
textShadowOffset	文字阴影位置	通用	设置为如下格式对象： {width:xx,height:xx}
textShadowRadius	文字阴影模糊半径	通用	数值
includeFontPadding	是否保留文字字体内间距	android	布尔值
textAlignVertical	文字竖直方向对齐模式	android	可选字符串： • auto：自动 • top：上对齐 • bottom：下对齐 • center：居中对齐
fontVariant	文字变体	iOS	数组，数组中的元素可选如下字符串： • small-caps • oldstyle-nums • lining-nums • tabular-nums • proportional-nums
letterSpacing	字符间距	iOS	数值
textDecorationColor	修饰线颜色	iOS	特定格式的 color 值
textDecorationStyle	修饰线风格	iOS	可选字符串： • solid：直线 • double：双线 • dotted：短虚线 • dashed：长虚线
writingDirection	书写方向	iOS	可选字符串： • auto：自动 • ltr：左方向 • rtl：右方向

表 6-1 中给出的文字风格可设置属性、相关释义及所设置值的相关信息都非常完善，建议使用前边编写的示例工程，将这些属性一一进行测试并观察效果。练习与实践是学好编程的不二法门！

6.1.2 Text 组件属性的设置

在 6.1.1 小节中我们介绍的其实只是 Text 组件的一个属性：style。style 属性专门用来设置组件的文字风格，除了 style 之外，Text 组件中还定义了许多属性，用来设置文字行数、组件交互效果等。在 React Native 中，组件属性的设置采用的是与 HTML 一致的语法规则。修改 TextDemo 类如下：

```
    export default class TextDemo extends Component {
        render(){
            return (
                //Text 控件
                <Text style={[style,base,iosStyle,androidStyle]}
adjustsFontSizeToFit = {false}
                numberOfLines = {2}
                onLayout = {({nativeEvent:{layout:{x,y,width,height}}})=>{
                    console.log(x,y,width,height);
                }}
                onLongPress={()=>{
                    console.log("长按");
                }}
                onPress={()=>{
                    console.log("按下");
                }}
                selectable={true}
                suppressHighlighting={true}
                >Hello Friend!Hello Friend!Hello Friend!Hello Friend!</Text>
            );
        }
    };
```

上面代码设置 Text 组件显示行数为 2 行，超出部分将会被截断。上面的 onLayout 属性、onLongPress 属性和 onPress 属性分别用于一些用户交互事件的监听。onLayout 用于监听布局变化或第一次挂载布局完成的消息，其中传入的参数格式如下：

```
    {nativeEvent: {layout: {x, y, width, height}}}
```

上面代码中我们使用了解构赋值来获取布局信息。onLongPress 方法会在 Text 组件被长按时调用，onPress 方法会在 Text 组件被单击时调用。

表 6-2 列出了 Text 组件的常用属性及释义。

表 6-2　Text 组件的常用属性

属性名	解释	平台	值类型或可选值
adjustsFontSizeToFit	设置文字是否自适应大小	通用	布尔值
allowFontScaling	设置文字是否根据系统字体大小进行缩放	通用	布尔值
numberOfLines	设置文本显示行数	通用	数值
onLayout	当组件布局变化或挂载完成后调用	通用	函数，其中参数格式如下：{nativeEvent: {layout: {x, y, width, height}}}
onLongPress	当组件被长按后调用此函数	通用	函数

（续表）

属性名	解释	平台	值类型或可选值
onPress	当组件被单击时调用此函数	通用	函数
selectable	设置是否允许进行长按复制文本	通用	布尔值
minimumFontScale	当允许文字尺寸自适应时，设置最小的文字缩放比例	iOS	数值在 0.01～1 之间
suppressHighlighting	设置是否显示交互效果	iOS	布尔值，如果设置为 true，则组件被按下时无效果，如果设置为 false，则组件被按下时会有阴影

6.1.3 Text 组件的嵌套

React Native 中的 Text 组件还有一个十分强大的特性，在于其支持嵌套。在 iOS 原生开发中，富文本的创建往往需要使用到 NSAttributeString 类，使用它创建的富文本编写复杂且不直观。在 React Native 中，你可以使用 Text 组件的嵌套来实现富文本渲染，例如：

```
export default class TextDemo extends Component {
    render(){
        return (
            //Text 控件
            <Text style={base} adjustsFontSizeToFit = {false}
            numberOfLines = {2}
            onLayout = {({nativeEvent:{layout:{x,y,width,height}}})=>{
                console.log(x,y,width,height);
            }}
            onLongPress={()=>{
                console.log("长按");
            }}
            onPress={()=>{
                console.log("按下");
            }}
            selectable={true}
            suppressHighlighting={true}
            >Hello Friend!
                <Text style={{color:'red',fontSize:25}}>Hello World</Text>
            </Text>
        );
    }
};
```

上面代码的运行效果如图 6-4 所示。

注意，在进行 Text 组件的嵌套时，内层的 Text 组件会默认继承外层 Text 组件的属性设置，设置内层 Text 组件的属性会覆盖掉继承效果。

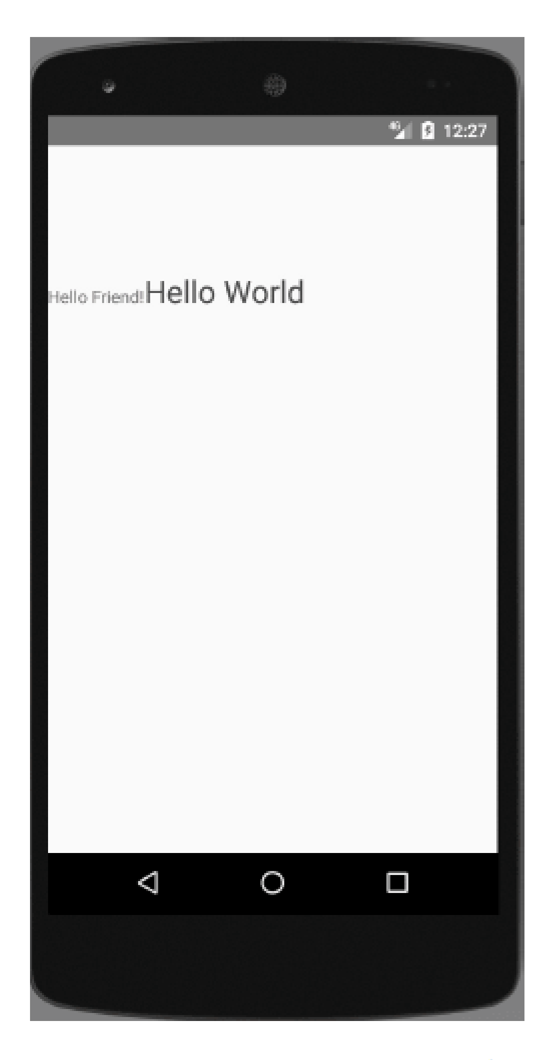


图 6-4　Text 组件的嵌套

6.1.4　React Native 程序的调试

前面的示例代码中我们使用了 console.log()方法进行打印操作，那么这些信息会输出在哪里，我们又如何来进行 React Native 程序的调试呢？其实也十分容易，React Native 应用程序支持在 Google Chrome 浏览器中进行远程调试。首先在模拟器中运行项目，在 iOS 模拟器中使用 coommand+d 可以调出菜单栏，在 Android 模拟器中则是使用 command+m 来调出菜单栏，iOS 模拟器与 Android 模拟器菜单栏效果分别如图 6-5 和图 6-6 所示。

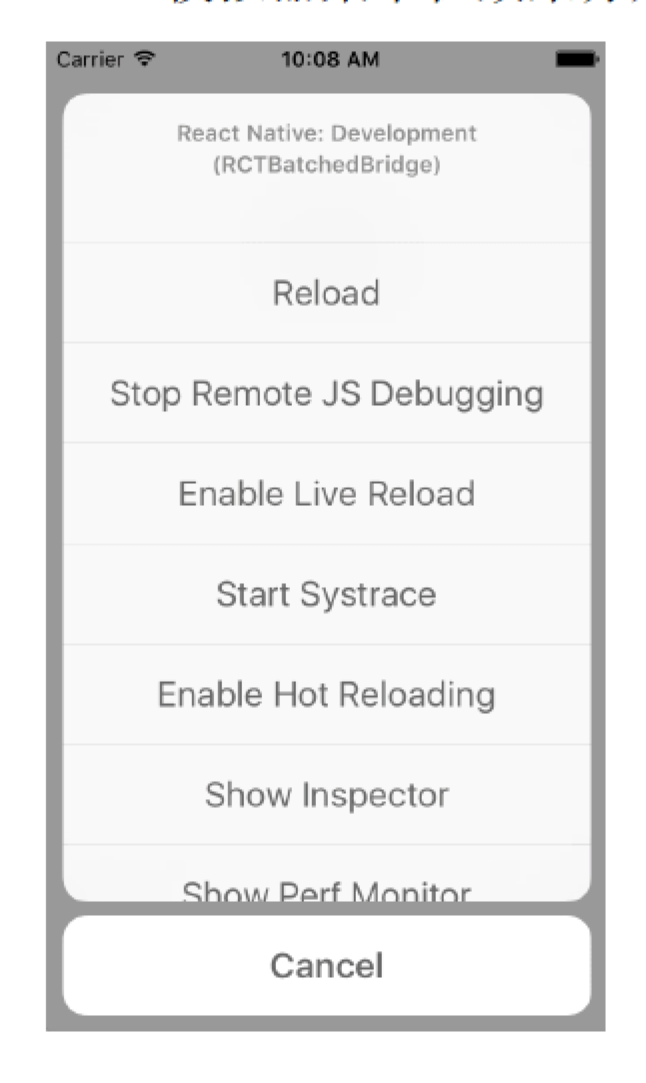



图 6-5　iOS 模拟器菜单栏

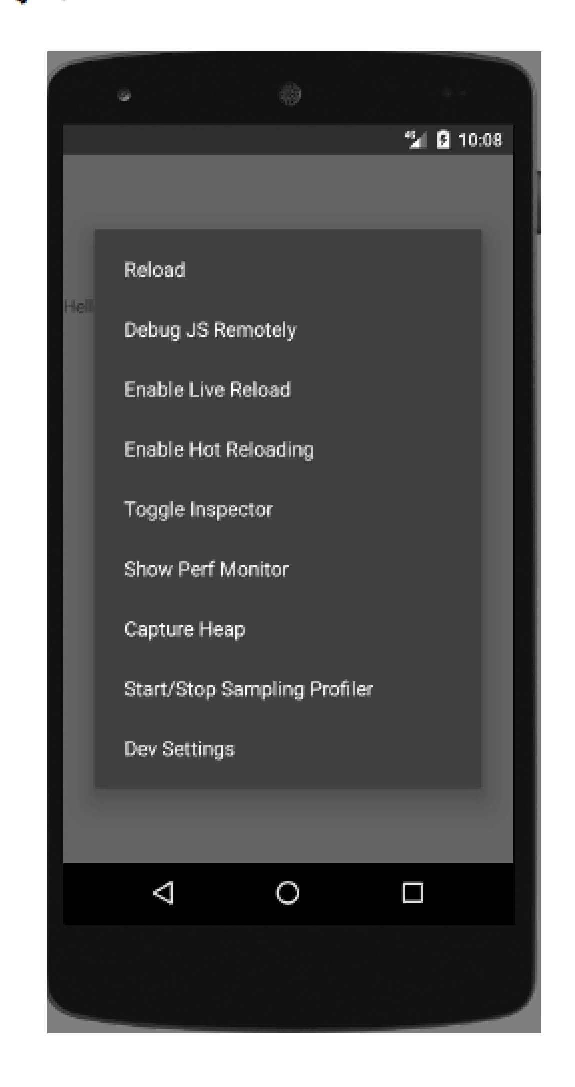


图 6-6　Android 模拟器菜单栏

单击菜单栏中的 Debug JS Remote 选项即可开启调试模式（如果已经开启调试模式，这里将显示 Stop Romote JS Debugging，用来关闭调试模式）。打开调试模式后会自动弹出 Google Chrom 浏览器的窗口，效果如图 6-7 所示。

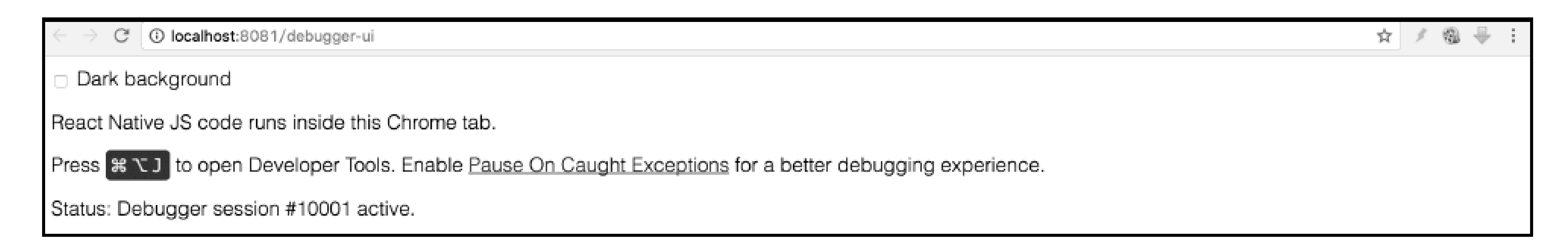


图 6-7　Google Chrom 浏览器的 React Native 调试界面

使用 command+option+j 即可打开浏览器的开发者模式，在其中可以看到我们的 JavaScript 代码以及控制台的打印信息，也可以方便地添加断点进行调试。

6.2　Button 按钮组件的应用

Button 组件是 React Native 中一个简单的按钮交互组件，其样式无法进行太多的定制化，对于复杂的交互控件，我们一般会采用 TouchableOpacity 组件或 TouchableNativeFeedback 组件来定制。

6.2.1　Button 组件的简单使用

在 HelloWorld 工程的 Demo 文件夹中新建一个命名为 ButtonDemo.js 的文件，在其中编写如下测试代码：

```
/*
按钮控件 Button
*/
import React,{Component} from 'react';
import {Button,View} from 'react-native';
export default class ButtomDemo extends Component{
    render(){
        return(
            <View style={{top:100}}>
                <Button title="按钮"
                color='red'
                onPress={()=>{
                    console.log("Button");
                }}
                disabled={false}/>
            </View>
        );
    }
}
```

Button 组件中可设置的属性非常少，并且无法定义 style 样式，因此如果我们需要对 Button 组件进行位置尺寸的控制，需要将其嵌套入一个 View 组件中。Button 组件支持设置的属性如表 6-3 所示。

表 6-3　Button 组件的属性

属性名	解释	平台	值类型或可选值
accessibilityLabel	读屏软件会读取这个属性的内容	通用	字符串
color	对于 iOS 平台，其设置文字的颜色；对于 Android 平台，其设置的是按钮的背景颜色	通用	特定格式的 color 值
disabled	设置此按钮是否有效	通用	布尔值 true：按钮可点击 false：按钮不可点击
onPress	设置按钮的触发方法	通用	函数
title	设置按钮的标题	通用	字符串

将 index.ios.js 和 index.Android.js 文件的内容修改如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
} from 'react-native';
import ButtonDemo from './Demo/ButtonDemo';
export default class HelloWorld extends Component {
  render() {
    return (<ButtonDemo />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

运行工程，两平台的效果分别如图 6-8 和图 6-9 所示。

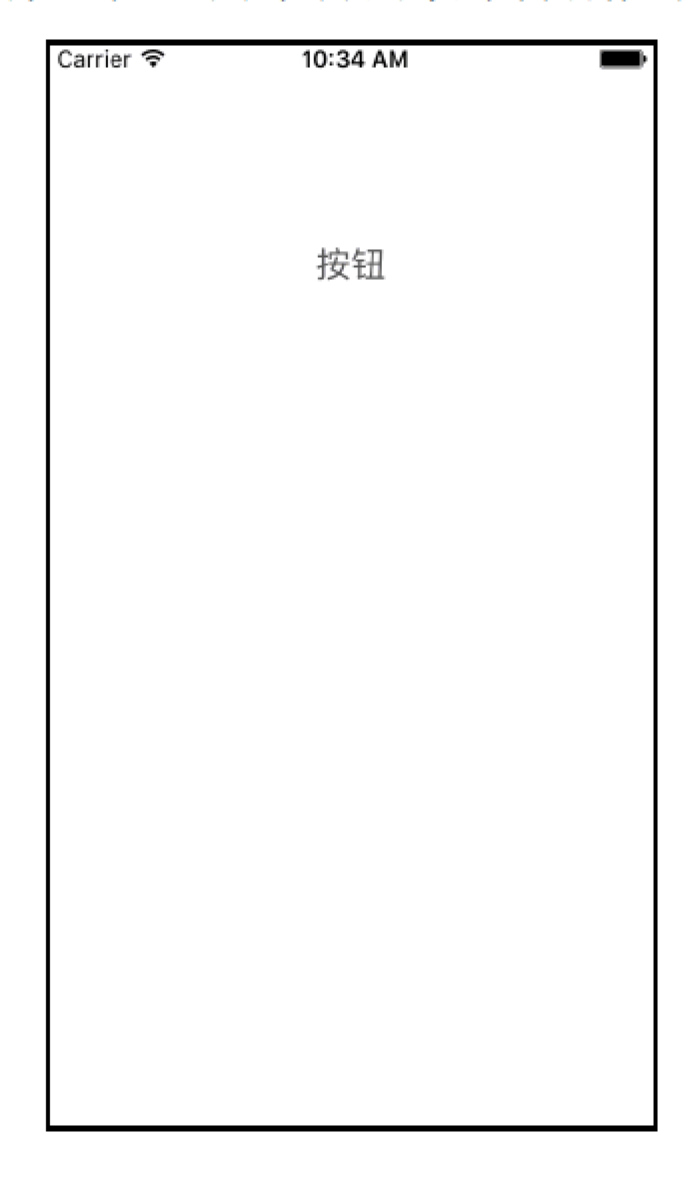




图 6-8　iOS 平台按钮效果



图 6-9　Android 平台按钮效果

6.2.2 小应用：屏幕霓虹灯

本小节我们来实现一个功能，当单击按钮的时候实现屏幕的背景色随机切换。在 ButtonDemo.js 文件中编写如下代码：

```
    import React,{Component} from 'react';
    import {Button,View} from 'react-native';
    export default class ButtomDemo extends Component{
        constructor(props){
            super(props);
            this.state = {
                style:{
                    backgroundColor:'blue',
                    flex:1
                }
            };
        }
        render(){
            return(
                <View style={this.state.style}>
                    <View style={{top:100}}>
                        <Button title="按钮"
                        color='red'
                        onPress={this.changeColor}
                        disabled={false}/>
                    </View>
                </View>
            );
        }
        changeColor=()=>{
            this.setState({
                style:{
        backgroundColor:'rgb('+(Math.random()*255)+','+(Math.random()*255)+','
+(Math.random()*255)+')',
                    flex:1
                }
                });
        }
    }
```

在上面的代码中，最外层的 View 组件充满整个设备屏幕，为背景容器，在对风格 style 进行设置时我们采用了 state 状态来控制。state 是 React Native 中组件用来保存属于本身的某些可变数据的属性。在 constructor 构造方法中需要对其进行初始化，当我们通过 setState()方法来重新修改这个属性时，会自动进行组件的重新渲染，起到动态刷新的作用。上面代码中有一个语法点需要格外

注意，如果 changeColor 是普通函数，则在调用时，其内部的 this 是指向 Button 组件的，并不是我们自定义的 ButtonDemo 组件，因此这里必须使用箭头函数，你一定还记得，箭头函数中的 this 是被固化的，其中 this 的指向在函数生成时就已经固定，始终指向 ButtonDemo 实例对象。

运行工程，连续单击按钮，屏幕向霓虹灯一样闪烁起来。

6.3　Image 图像组件的应用

Image 组件用于在界面上渲染图像，其所渲染的图像可以是 React Native 工程中的本地图片素材、base64 编码的图像字符串、iOS 或 Android 原生工程中的图像素材以及网络素材。在 React Native 中，Image 是一个十分强大的 UI 组件。

6.3.1　渲染图像的方式

React Native 提供了统一的方式来管理 iOS 与 Android 项目中的图像素材。如果需要在界面中显示一个本地的图像资源，最简单的方式是将图片文件放入 React Native 工程中，例如我们在 HelloWorld 工程中新建一个文件夹，命名为 source，此时 HelloWorld 工程结构应该如图 6-10 所示。

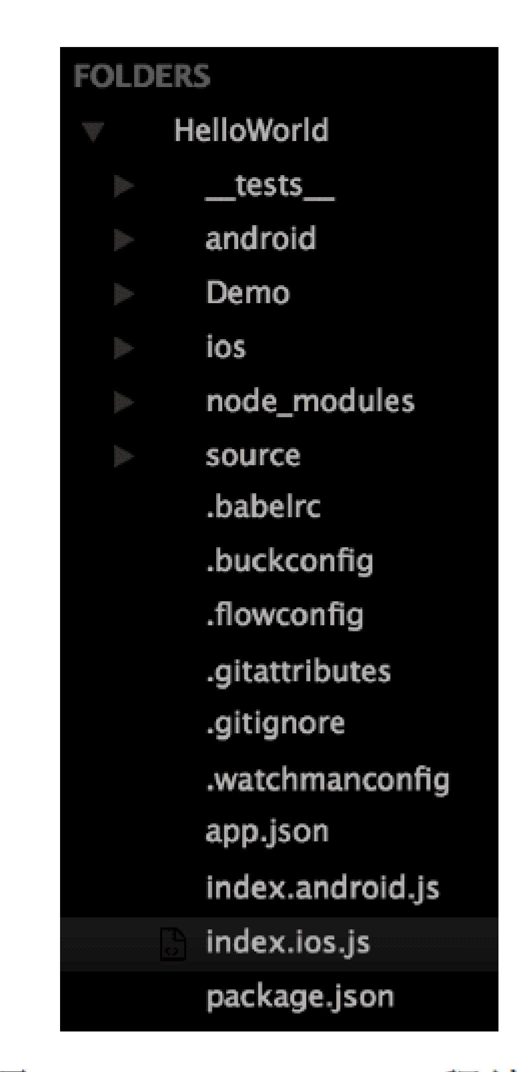


图 6-10　HelloWorld 工程结构

在 source 文件夹下放入一张 png 图像文件作为测试图片。在 Demo 文件夹下新建一个文件，命名为 ImageDemo.js。在其中编写如下测试代码：

```
import React,{Component} from 'react';
import {Image} from 'react-native';
export default class ImageDemo extends Component{
    render(){
        return(
            <Image source={require('../source/image.png')}/>
        );
    }
}
```

Image 组件的 source 属性设置要渲染图片的来源，如果是本地静态图片，可以使用 require() 方法来引入，其中需要设置正确的图片文件相对路径。将 index.ios.js 与 index.Android.js 文件修改如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
```

```
} from 'react-native';
import ImageDemo from './Demo/ImageDemo';
export default class HelloWorld extends Component {
  render() {
    return (<ImageDemo />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

在双平台上运行项目，效果如图 6-11 所示。

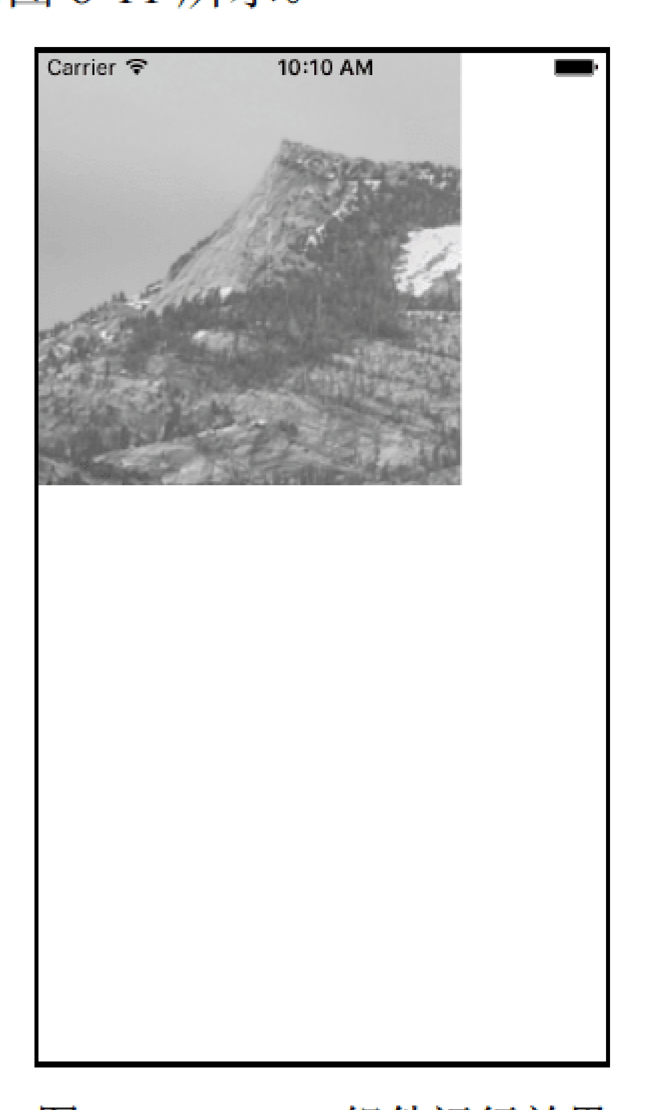


图 6-11　Image 组件运行效果

抽 丝 剥 茧

注意，在进行本地图像素材检索的时候，你可以根据不同的屏幕分辨率提供不同的素材，只需要为图片素材添加@2x、@3x 这样的后缀即可。同样，你也可以使用同一个图像素材名称为 iOS 平台和 Android 平台提供不同的素材文件，需要遵守 image.ios.png (image.android.png)这样的素材命名方式。在使用时，直接使用 image.png 即可，React Native 会自动根据平台选择正确的图片进行打包。

使用 React Native 做混合开发也是一种不错的选择。所谓混合开发，是指应用程序中有些部分是 React Native 开发的，有些部分是原生开发的。在这种情况下难免会有素材共用的场景，在 React Native 中也可以直接使用 Xcode 工程 xcassets 目录或者 Android 工程的 drawable 目录中的素材。示例代码如下：

```
render(){
        return(
            <Image style={{width:100,height:50}} source={{uri: 'image'}}/>
        );
    }
```

这种方式加载本地图片必须指定图片的尺寸，uri 属性直接指定图像文件的名称，不需要带后缀。需要注意，对于 Android 工程，文件素材必须添加到 res 目录下的 drawable 文件夹里，并且重新编译。对于 iOS 工程，图像文件则必须放在 xcassets 目录中，并且需要在 Xcode 中重新编译项目。

Image 组件也可以直接加载网络图片，示例如下：

```
        render(){
            return(
                <Image source={{uri: 'https://facebook.github.io/react/img/
ogo_og.png'}}
            style={{width: 200, height: 200}} />
            );
        }
```

注意，这种方式加载的图片也必须指定好尺寸。

除了上面说的几种方式外，Image 组件也可以直接显示 base64 编码的字符串图片，示例如下：

```
    var base64Icon = 'data:image/png;
base64,iVBORw0KGgoAAAANSUhEUgAAAEsAAABLCAQAAACSR7JhAAADtUlEQVR4Ac3YA2Bj6QLH0XP
T1Fzbtm29tW3btm3bfLZtv7e2ObZnms7d8Uw098tuetPzrxv8wiISrtVudrG2JXQZ4VOv+qUfmqCGG
l1mqLhoA52oZlb0mrjsnhKpgeUNEs91Z0pd1kvihA3ULGVHiQO2narKSHKkEMulm9VgUyE60s1aWoM
QUbpZOWE+kaqs4eLEjdIlZTcFZB0ndc1+lhB1lZrIuk5P2aib1NBpZaL+JaOGIt0ls47SKzLC7Cqrl
GF6RZ09HGoNy1lYl2aRSWL5GuzqWU1KafRdoRp0iOQEiDzgZPnG6DbldcomadViflnl/cL93tOoVbs
OLVM2jylvdWjXolWX1hmfZbGR/wjypDjFLSZIRov09BgYmtUqPQPlQrPapecLgTIy0jMgPKtTeob2z
WtrGH3xvjUkPCtNg/tm1rjwrMa+mdUkPd3hWbH0jArPGiU9ufCsNNWFZ40wpwn+62/66R2RUtoso1O
B34tnLOcy7YB1fUdc9e0q3yru8PGM773vXsuZ5YIZX+5xmHwHGVvlrGPN6ZSiP1smOsMMde40wKv2V
mwPPVXNut4sVpUreZiLBHi0qln/VQeI/LTMYXpsJtFiclUN+5HVZazim+Ky+7sAvxWnvjXrJFneVtL
WLyPJu9K3cXLWeOlbMTlrIelbMDlrLenrjEQOtIF+fuI9xRp9ZBFp6+b6WT8RrxEpdK64BuvHgDk+v
Uy+b5hYk6zfyfs051gRoNO1usU12WWRWL73/MMEy9pMi9qIrR4ZpV16Rrvduxazmy1FSvuFXRkqTnE
7m2kdb5U8xGjLw/spRr1uTov4uOgQE+0N/DvFrG/Jt7i/FzwxbA9kDanhf2w+t4V97G8lrT7wc08aA
2QNUkuTfW/KimT01wdlfK4yEw030VfT0RtZbzjeMprNq8m8tnSTASrTLti64oBNdpmMQm0eEwvfPwR
bUBywG5TzjPCsdwk3IeAXjQblLCoXnDVeoAz6SfJNk5TTzytCNZk/POtTSV40NwOFWzw86wNJRpubp
Xsn60NJFlHeqlYRbslqZm2jnEZ3qcSKgm0kTli3zZVS7y/iivZTweYXJ26Y+RTbV1zh3hYkgyFGSTK
PfRVbRqWWVReaxYeSLarYv1Qqsmh1s95S7G+eEWK0f3jYKTbV6bOwepjfhtafsvUsqrQvrGC8YhmnO
9cSCk3yuY984F1vesdHYhWJ5FvASlacshUsajFt2mUM9pqzvKGcyNJW0arTKN1GGGzQlH0tXwLDgQT
urS8eIQAAAABJRU5ErkJggg==';
    export default class ImageDemo extends Component{
        render(){
            return(
                <Image source={{uri: base64Icon}}
            style={{width: 200, height: 200}} />
            );
        }
    }
```

运行工程，可以看到图像也被渲染到了模拟器上。

6.3.2 Image 组件的风格自定义

Image 组件支持对 style 属性进行设置，Image 组件所支持的风格设置有 4 类，分别是基本视图类、阴影类、动画变换类和 Image 组件定制类。其中，基本视图类用来设置一些布局属性，后面会有专题进行介绍。阴影类的属性只在 iOS 平台有效。动画变换类的风格属性用来对 Image 组件进行旋转、缩放、平移等操作。Image 组件定制类则可以对 Image 组件进行背景色、圆角、边框等处理。在 ImageDemo.js 中编写如下示例代码：

```
    import React,{Component} from 'react';
    import {Image} from 'react-native';
    export default class ImageDemo extends Component{
        render(){
            return(
                <Image source={require('../source/image.png')}
                style={[shadowStyle,baseStyle,imageStyle]} />
            );
        }
    }
    //base
    let baseStyle = {
        width:150,
        height:150,
    }
    //transform
    //transform
    let tansformStyle = {
        transform:[{rotate:'0.1rad'},{rotateX: '0.1rad'}, {rotateY: '60deg'},
{rotateZ: '0.1rad'}, {scale: 0.9}, {scaleX: 0.9}, {scaleY: 0.9}, {translateX: 50},
{translateY: 50}, {skewX: '20deg'}, {skewY: '20deg'}]
    }//shadow
    let shadowStyle = {
        shadowColor:'red',
        shadowOffset:{
            width:5,
            height:5
        },
        shadowOpacity:1,
        shadowRadius:5
    }
    //image
    let imageStyle = {
        backfaceVisibility:'visible',
        overflow:'visible',
```

```
        resizeMode:'contain',
        backgroundColor:'green',
        borderColor:'blue',
        borderWidth:3,
        //指定所有非透明的颜色为某个颜色
        // tintColor:'purple',
        opacity:1,
        //android
        borderBottomLeftRadius:30,
        borderBottomRightRadius:30,
        overlayColor:'black'
    }
```

注意，与 Image 阴影相关的属性设置只在 iOS 平台有效，并且必须将 overflow 属性设置为 visible 时有效。阴影相关属性如表 6-4 所示。

表 6-4　阴影相关属性

属性名	解释	平台	值类型或可选值
shadowColor	设置阴影颜色	iOS	特定的颜色字符串
shadowOffset	设置阴影位置偏移	iOS	如下格式的对象： {width:number,height:number}
shadowOpacity	设置阴影的透明度	iOS	数值（0～1）
shadowRadius	设置阴影的模糊半径	iOS	数值

动画变换部分的属性用来对 Image 组件进行一些空间上的修改，属性列表如表 6-5 所示。

表 6-5　动画变换属性

属性名	解释	平台	值类型或可选值
transform	进行变换设置	通用	特定格式对象

我们需要具体讲一讲 transform 这个属性的应用。其对象基本格式如下：

```
    //transform
    let tansformStyle = {
        transform:[{rotate:'0.1rad'},{rotateX: '0.1rad'}, {rotateY: '60deg'},
{rotateZ: '0.1rad'}, {scale: 0.9}, {scaleX: 0.9}, {scaleY: 0.9}, {translateX: 50},
{translateY: 50}, {skewX: '20deg'}, {skewY: '20deg'}]
    }
```

其中，rotate 表示进行旋转变换，可以用角度制与弧度制两种表达方式，以 rad 为后缀表示使用弧度制，以 deg 为后缀表示使用角度制，rotateX、rotateY、rotateZ 分别设置相对 X 轴、Y 轴、Z 轴的旋转变换。scale 表示进行缩放变换，scaleX 和 scaleY 分别表示相对 X 轴和 Y 轴进行缩放。translateX 和 translateY 用于平移变换，分别表示沿 X 轴方向平移和沿 Y 轴方向平移。skew 则是用于进行斜切变换，skewX 和 skewY 分别表示相对 X 轴或 Y 轴斜切。

用于定制 Image 组件风格的属性如表 6-6 所示。

表 6-6 Image 组件风格属性

属性名	解释	平台	值类型或可选值
backfaceVisibility	设置图片背面是否可见（在旋转变化中会有效果）	通用	可选字符串： • visible：背面可见 • hidden：背面不可见
resizeMode	设置图片拉伸模式	通用	可选字符串： • cover：等比拉伸 • strech：保持原有代销 • contain：拉伸到充满空间
backgroundColor	设置容器背景颜色	通用	特定的颜色字符串
borderColor	设置组件边框颜色	通用	特定的颜色字符串
borderRadius	设置组件边框圆角半径	通用	数值
borderWidth	设置组件边框宽度	通用	数值
overflow	设置超出边框的部分是否可见	通用	可选字符串： • visible：可见 • hidden：不可见
tintColor	为所有非透明像素指定颜色	通用	特定的颜色字符串
opacity	设置组件透明度	通用	数值（0～1）
overlayColor	图片圆角时指定颜色填充空白	Android	特定的颜色字符串
borderBottomLeftRadius	设置组件左下角圆角半径	Android	数值
borderBottomRightRadius	设置组件右下角圆角半径	Android	数值
borderTopLeftRadius	设置组件左上角圆角半径	Android	数值
borderTopRightRadius	设置组件右上角圆角半径	Android	数值

注意，对于 iOS 平台，圆角和阴影是不能同时设置的，当 overflow 属性设置为 visible 时，组件就无法再显示圆角，但是可以显示阴影，如果将 overflow 属性设置为 hidden，则阴影将会隐藏，圆角效果可以显示。运行工程，效果如图 6-12 所示。

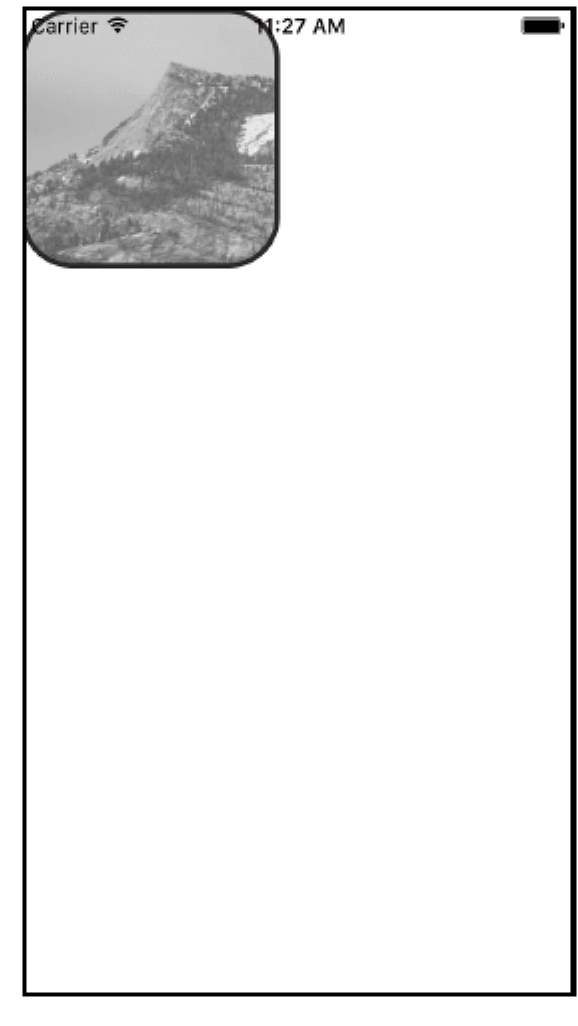


图 6-12　对 Image 组件进行风格定制

6.3.3　Image 组件的属性和方法解析

Image 组件既可以加载本地的图像素材，也可以加载网络上的图像素材。Image 组件中还提供了图片加载过程的监听，修改 ImageDemo 类的 render 方法如下：

```
render(){
    return(
        <Image source={require('../source/image.png')}
        style={[shadowStyle,baseStyle,imageStyle]}
        onLayout={()=>{
            console.log("onLayout");
        }}
        onLoad={()=>{
            console.log("onLoad");
        }}
        onLoadEnd={()=>{
            console.log("onLoadEnd");
        }}
        onLoadStart={()=>{
            console.log("onLoadStart");
        }}
        />
    );
}
```

在模拟器中运行工程，打开调试模式，输出面板效果如图 6-13 所示。

```
onLayout                                    ImageDemo.js:10
onLoadStart                                 ImageDemo.js:19
onLoad                                      ImageDemo.js:13
onLoadEnd                                   ImageDemo.js:16
```

图 6-13　控制台输出的打印信息

实际上，onLoad 属性设置的回调函数与图像的加载无关，当 Image 组件挂载完成或布局改变的时候会被调用。onLaodStart 属性设置的回调函数当组件将要开始加载图片时调用。onLaod 属性设置的回调函数在组件加载完成时会回调。onLaodEnd 属性设置的回调函数当组件加载结束后会回调，这个函数无论图像加载成功或失败都会被回调。

Image 组件支持设置的属性列表如表 6-7 所示。

表 6-7　Image 组件支持设置的属性

属性名	解释	平台	值类型或可选值
onLaod	组件挂载完成或布局改变时调用的回调	通用	函数
onLoad	组件加载成功后调用的回调	通用	函数

（续表）

属性名	解释	平台	值类型或可选值
onLoadEnd	组件加载结束后调用的回调	通用	函数
onLoadStart	组件开始加载时调用的回调	通用	函数
resizeMode	设置图像的拉伸模式	通用	可选字符串： • cover：保持宽高比不变将图片充满整个容器 • contain：保持宽高比不变将图片缩放到刚好适应容器 • stretch：拉伸图片到充满容器 • repeat：保持图片原有尺寸进行赋值平铺，仅 iOS 平台可用 • center：居中不拉伸
source	设置图像资源	通用	带有 uri 属性的对象或者 require 引入的路径
resizeMethod	设置图像的拉伸方法	Android	可选字符串： • auto：自动计算 • resize：在图片解码之前对内存中的数据进行修改 • scale：对图片进行缩放
accessibilityLabel	读屏软件会读取这个属性的值	iOS	字符串
blurRadius	为图像添加模糊滤镜，设置模糊半径	iOS	数值
capInsets	当图像拉伸的时候，capInsets 指定的边缘不会被拉伸	iOS	特定对象： {top:number,left:number, bottom:number,right:munber}
defaultSource	在读取图片时默认加载的图片	iOS	特定对象： {uri:string,width:number, height:number,scale:number}
onError	当加载图片出错的时候调用的回调	iOS	函数
onPartialLoad	如果图片支持分步加载，在分步加载过程中会被调用	iOS	函数
onProgress	在加载过程中不断调用，返回图片的加载进度	iOS	特定参数的函数，参数格式示例如下： onProgress={({ nativeEvent: {loaded, total}})=> {console.log(loaded,total); }}

除了上面所提到的属性，Image 组件中还定义了两个静态方法，如表 6-8 所示。

表 6-8　Image 组件的两个静态方法

方法名	解释	平台	参数
getSize(uri,success,error)	获取图像的尺寸	通用	• success：函数，参数格式为(width: number, height: number)，回调时可以获取图像的尺寸 • error：函数，参数格式为(error: any)
perfetch(uri)	预加载一个远程图片	通用	略

其中，getSize 方法可以预先获取一个图像素材的尺寸，示例如下：

```
        render(){
            Image.getSize('http://facebook.github.io/react/img/logo_og.png',
(width, height)=>{
                        console.log(width,height);
                        }, (any)=>{
                        console.log(any);
                        });
            return(
                <Image source={require('../source/myImage.png')} style=
{[baseStyle,shadowStyle,imageStyle,transfromStyle]}
                onLayout={()=>{
                    console.log("onLayout");
                }}
                onLoad={()=>{
                    console.log("onLoad");
                }}
                onLoadEnd={()=>{
                    console.log("onLoadEnd");
                }}
                onLoadStart={()=>{
                    console.log("onLoadStart");
                }}
                blurRadius={10}
                />

                );
        }
```

抽 丝 剥 茧

这里讲的静态方法是指定义在 Image 类上面的方法，使用 Image 类来进行调用，定义在实例对象或者原型链上的方法称为示例方法。

Image 组件可以以单标签的形式作为独立组件，也可以以双便签的形式来嵌套其他组件，它的这种特性常用于为其他组件添加背景图，例如：

```
    render(){
            return(
                <Image
                source={{uri:"http://facebook.github.io/react/img/
logo_og.png"}}
                defaultSource={require('../source/timg.jpeg')}
                style={[shadowStyle,baseStyle,imageStyle]}
                blurRadius={10}
                >
                    <Text style={{top:50,color:"white",backgroundColor:
'#00000000'}}
                    >
                        你好，Image!
                    </Text>
                </Image>
            );
        }
```

运行工程，效果如图 6-14 所示。

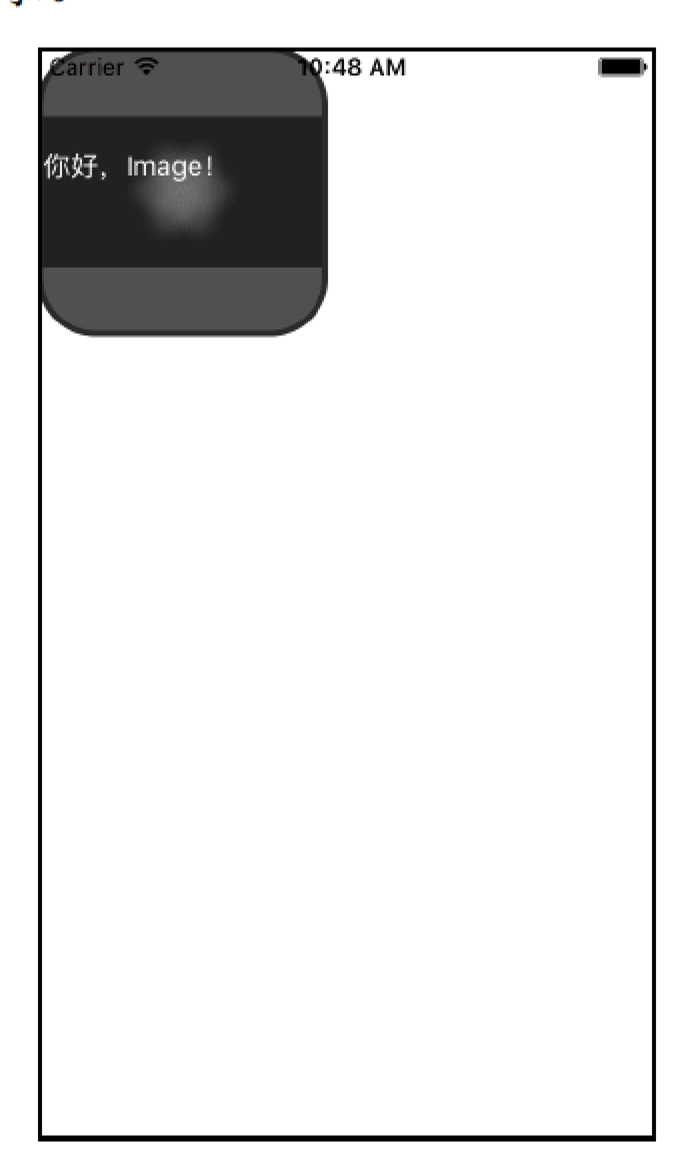


图 6-14　为 Text 组件添加背景图

6.4　Switch 开关组件的应用

Switch组件用于创建一个开关控件，其只有两种状态，分别为开或者关。在iOS平台中有UISwitch这样一个原生的开发组件，Android 平台并不支持，因此此组件在两个平台的样式差异较大。在HelloWorld 工程的 Demo 文件夹中新建一个命名为 SwitchDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {Switch,View} from 'react-native';
export default class SwitchDemo extends Component{
    constructor(props){
        super(props);
        this.state = {
            value:false
        }
    }
    render(){
        return(
            <Switch style={switchStyle}
            value={this.state.value}
            disabled={false}
            onValueChange={
            (value)=>{
                this.setState({value:value});
                console.log(value);
            }}
            onTintColor='red'
            thumbTintColor='blue'
            tintColor='green'/>
        );
    }
}
let switchStyle = {
    top:100,
    left:100,
    width:50,
    height:50
}
```

注意，Switch 是一种受控组件，即当用户点击开关进行开关状态的切换时，首先会调用 onValueChange 属性设置的回调，在回调中会将开发的状态反馈给开发者，开发者需要手动调整开关的显示样式，使用 setState 方法来进行界面的刷新。运行工程，iOS 平台与 Android 平台效果分别如图 6-15、图 6-16 所示。

Switch 组件可设置的属性如表 6-9 所示。

表 6-9　Switch 组件可设置的属性

属性名	解释	平台	值类型或可选值
disabled	是否有效	通用	布尔值 • ture：表示可进行用户交互 • false：表示不可进行用户交互
onValueChange	当用户操作开关时调用的回调	通用	函数，会将开发状态作为参数传递

（续表）

属性名	解释	平台	值类型或可选值
value	组件状态	通用	布尔值 • true：开状态 • false：关状态
onTintColor	开启状态时的背景颜色	通用	特定的颜色字符串
thumbTintColor	开关滑块的颜色	通用	特定的颜色字符串
tintColor	关闭时的渲染颜色	通用	特定的颜色字符串

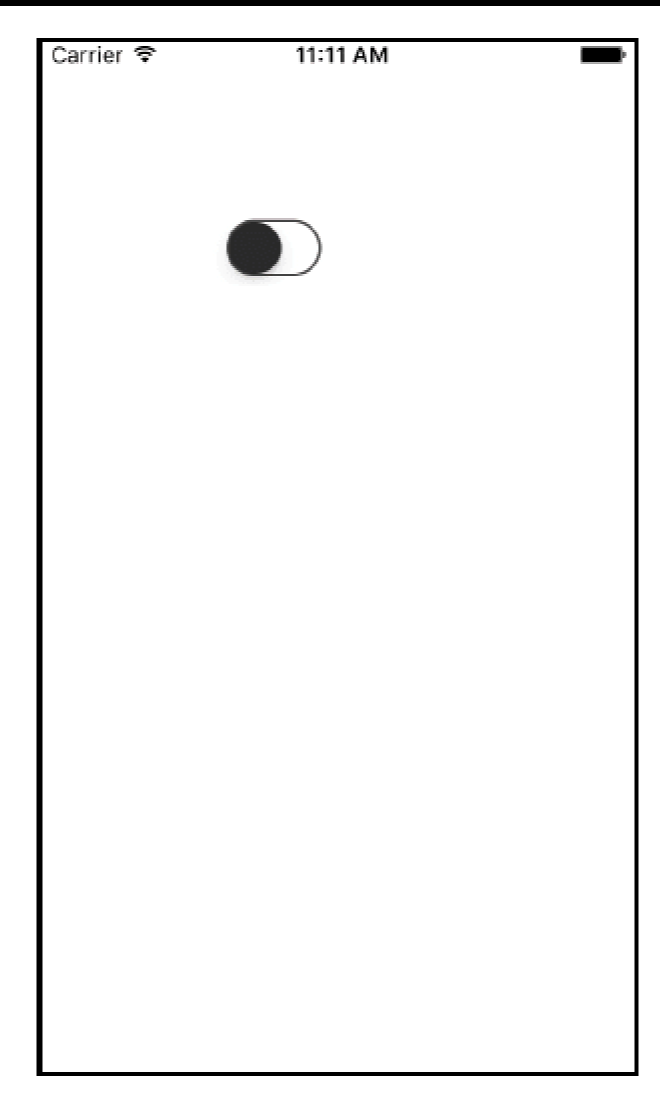


图 6-15　iOS 平台的 Switch 组件

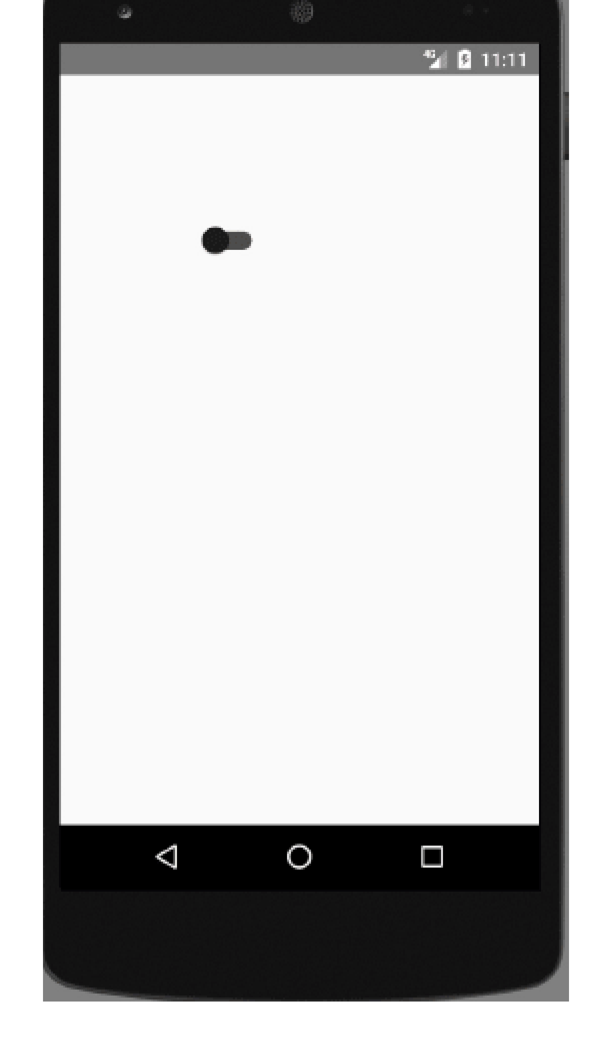


图 6-16　Android 平台的 Switch 组件

6.5　Slider 滑块组件的应用

滑块组件是 React Native 中提供的一个简洁的选值组件，对 iOS 与 Android 平台都有良好的支持。打开 HelloWorld 项目，在 Demo 文件夹下新建一个命名为 SliderDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {Slider} from 'react-native';
export default class SliderDemo extends Component{
    render(){
        return(
            <Slider style={baseStyle}/>
        );
    }
}
```

```
let baseStyle = {
    top:100,
    marginLeft:30,
    marginRight:30
}
```

上面创建了一个最简单的滑块组件，将其固定在其父容器距离左右各 30 个距离单位、上边 100 个距离单位的位置。修改 index.ios.js 与 index.Android.js 文件如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
} from 'react-native';
import SliderDemo from './Demo/SliderDemo';
export default class HelloWorld extends Component {
  render() {
    return (<SliderDemo />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

运行工程，效果如图 6-17 与图 6-18 所示。

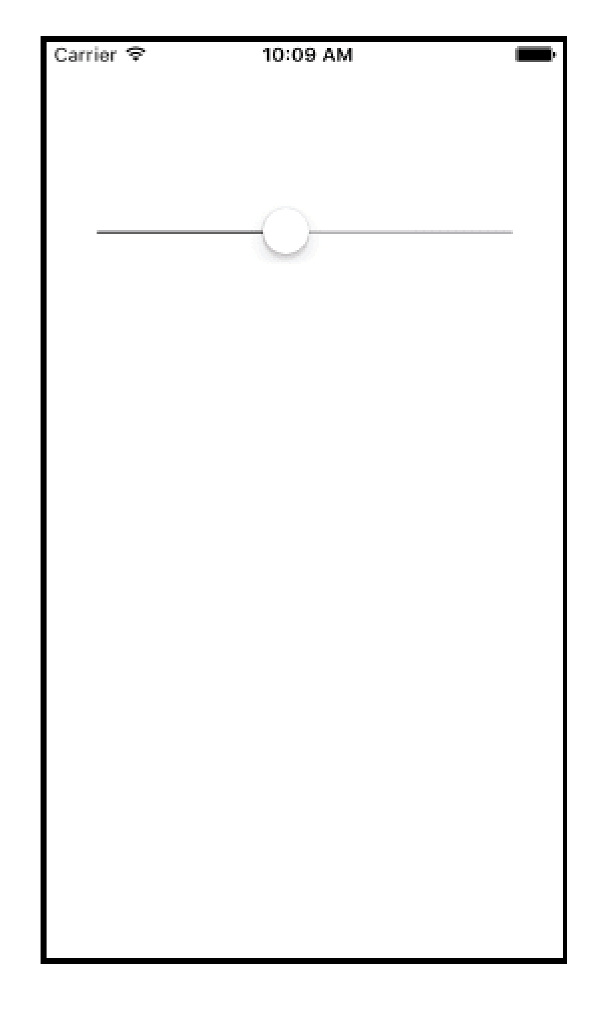



图 6-17　iOS 平台的滑块组件

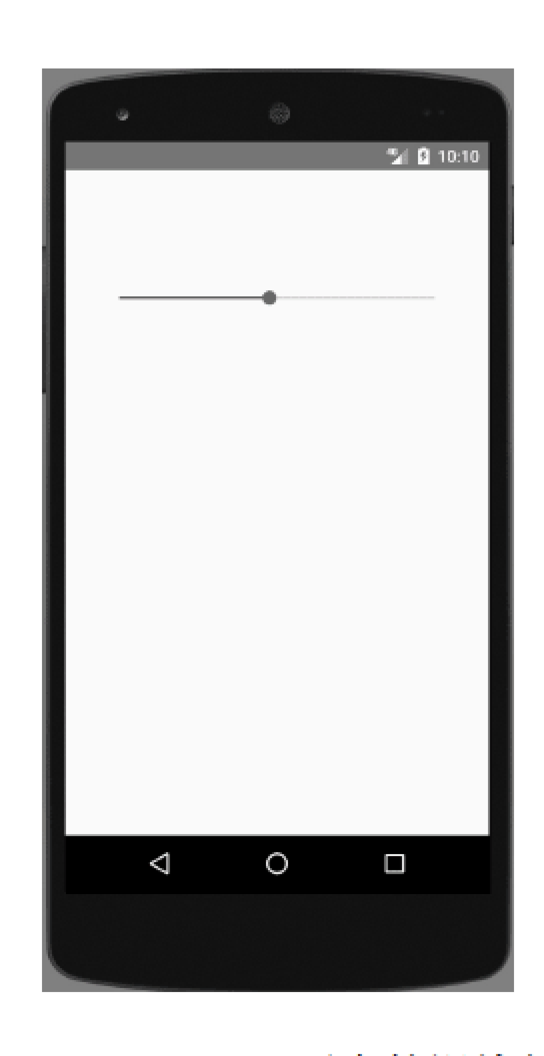


图 6-18　Android 平台的滑块组件

抽 丝 剥 茧

在 iOS 原生开发中，系统提供了 UISlider 这样一个滑块控件，因此 Slider 组件在 iOS 和 Android 平台上的表现有略微不同。

Slider 组件也支持样式和功能上的定制，其中提供了许多属性供我们直接使用，示例如下：

```
render(){
        return(
```

```
            <Slider style={baseStyle}
            disabled={false}
            maximumTrackTintColor="red"
            minimumTrackTintColor="blue"
            onSlidingComplete={(value)=>{
                console.log("slider finish",value);
            }}
            onValueChange={(value)=>{
                console.log("slider change",value);
            }}
            step={0.1}/>
        );
    }
```

Slider 组件支持的属性如表 6-10 所示。

表 6-10　Slider 组件可设置的属性

属性名	解释	平台	值类型或可选值
disabled	设置组件是否可用	通用	布尔值 • true：组件不可交互 • false：组件可交互
maxmumTrackTintColor	设置 iOS 平台轨道右侧颜色或 Android 平台轨道左侧颜色	通用	特定格式的颜色字符串
minimumTrackTintColor	设置 iOS 平台轨道左侧的颜色或 Android 平台轨道右侧颜色	通用	特定格式的颜色字符串
onSlidingComplete	用户滑动结束后会回调此属性设置的函数	通用	函数，会将 Slider 组件当前的值作为参数传递进来
onValueChange	当滑块组件的值改变时调用的回调	通用	函数，会将 Slider 组件当前的值作为参数传递进来
step	设置最小步长	通用	数值
thumbImage	设置滑块图片	通用	静态图片
trackImage	设置轨道图片	通用	静态图片
value	设置滑块的初始值	通用	数值
maximumTrackImag	设置滑块右侧轨道的背景图	iOS	静态图片
minimumTrackImage	设置滑块左侧轨道的背景图	iOS	静态图片
maximumValue	设置滑块的最大值	iOS	数值，默认为 1
minimumValue	设置滑块的最小值	iOS	数值，默认为 0
thumTintColor	设置滑块的背景颜色	Android	特定格式的颜色字符串

6.6　ActivityIndicator 指示器组件的应用

在体验优秀的应用程序中，我们经常会看到各式各样的指示器，例如在某些下载任务、数据存储或加载任务执行时，界面上都会显示一些等待特效。在 React Native 中提供了 ActivityIndicator 跨平台组件来创建活动指示器。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ActivityIndicatorDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {ActivityIndicator} from 'react-native';
export default class ActivityIndicatorDemo extends Component{
    render(){
        return(
            <ActivityIndicator style={baseStyle}
            animating={true}
            color="red"
            size='large'/>
        );
    }
}
let baseStyle = {
    marginHorizontal:0,
    top:100
}
```

修改 index.ios.js 与 index.Android.js 文件如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
} from 'react-native';
import ActivityIndicatorDemo from './Demo/ActivityIndicatorDemo';
export default class HelloWorld extends Component {
  render() {
    return (<ActivityIndicatorDemo />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

运行工程，可以看到在 iOS 平台与 Android 平台中 ActivityIndicator 的表现略有不同。

ActivityIndicator 组件支持的属性如表 6-11 所示。

表 6-11　ActivityIndicator 组件支持的属性

属性名	解释	平台	值类型或可选值
animating	开启动画	通用	布尔值
color	设置指示器的颜色	通用	特定格式的颜色字符串
size	设置指示器的尺寸	通用	特定的字符串 • small：小指示器 • large：大指示器
hidesWhenStopped	设置没有动画的时候是否自动隐藏指示器（Android 平台自动隐藏）	iOS	布尔值

6.7　TextInput 用户输入组件的应用

用户输入也是交互的一种，这种场景在实际开发中也经常会使用到。例如，在登录注册相关的界面免不了需要用户输入一些信息。TextInput 便是专门用来处理用户输入的组件。在 HelloWorld 工程的 Demo 文件夹下面新建一个命名为 TextInputDemo.js 的文件，在其中编写如下测试代码：

```
import React,{Component} from 'react';
import {TextInput} from 'react-native';
export default class TextInputDemo extends Component{
    render(){
        return(
            <TextInput style={baseStyle}
            />
        );
    }
}
let baseStyle = {
    top:100,
    marginLeft:30,
    marginRight:30,
    height:100,
    borderWidth:1,
    borderColor:'gray'
}
```

修改 index.ios.js 文件与 index.Android.js 文件如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
} from 'react-native';
```

```
import TextInputDemo from './Demo/TextInputDemo';
export default class HelloWorld extends Component {
  render() {
    return (<TextInputDemo />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

上面代码创建了一个简单的文本输入框，默认的 TextInput 是单行输入的，并且其 style 属性可定义的风格与 Text 组件完全一致。需要注意，在 iOS 平台上的 TextInput 组件默认无任何边框，在 Android 平台上 TextInput 组件会默认带底线，可以通过相关属性的设置来去掉这些差异。

运行工程，效果如图 6-19 所示。

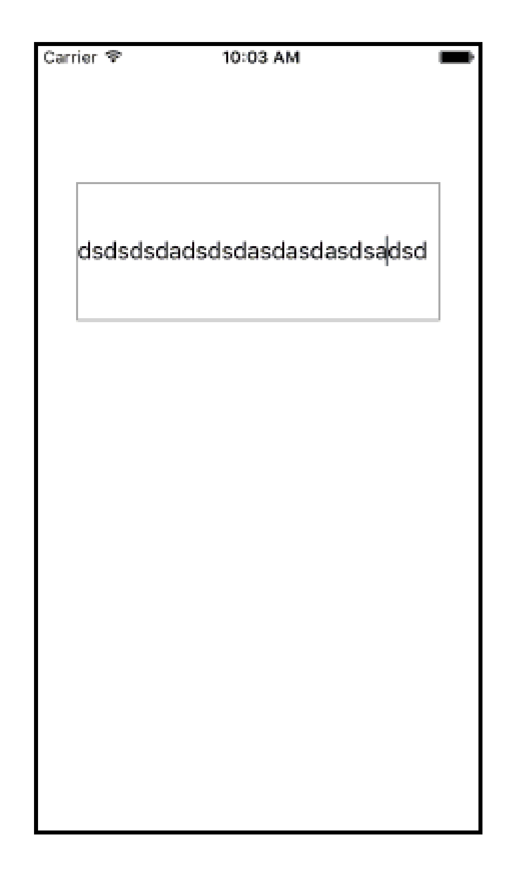


图 6-19　TextInput 组件

TextInput 组件中可以进行配置的属性如表 6-12 所示。

表 6-12　TextInput 组件的属性

属性名	解释	平台	值类型或可选值
autoCapitalize	设置 TextInput 指定字符自动大写	通用	特定枚举字符串 • characters：所有字符转换大写 • words：单词首字母大写 • sentences：句首大写 • none：不自动转换大写
autoCorrect	是否开启自动拼写检查	通用	布尔值 • true：开启，默认为 true • false：不开启
autoFocus	是否自动获取焦点	通用	布尔值 • true：自动获取 • false：不自动获取
blurOnSubmit	设置是否在回车提交时失焦，单行输入组件默认是，多行输入组件默认否	通用	布尔值 • true：回车失焦 • false：回车不失焦

（续表）

属性名	解释	平台	值类型或可选值
caretHidden	设置是否隐藏光标（目前只在 iOS 平台有效）	通用	布尔值 • true：隐藏光标 • false：不隐藏
defaultValue	提供文本输入组件的初始值	通用	字符串
editable	设置文本输入组件是否可以编辑	通用	布尔值 • true：可编辑 • false：不可编辑
keyboardType	设置弹出键盘的类型	通用	枚举字符串 • default：默认键盘 • numeric：数组键盘 • email-address：邮件键盘
maxLength	设置文本框最多可输入的字符数	通用	数值
multiline	设置是否支持多行文本	通用	布尔值 • true：支持多行文本 • false：单行文本
onBlur	设置当文本输入组件失去焦点时调用的回调	通用	函数
onChange	设置当文本输入组件变化时调用的回调	通用	函数
onChangeText	设置当文本输入组件内容变化时调用的回调	通用	函数，会将改变后的文字内容作为参数传递进来
onEndEditing	设置当文本输入结束后调用的回调	通用	函数
onFocus	设置当文本输入组件获得焦点时调用的回调	通用	函数
onLayout	当组件挂载完成或者布局改变时调用的回调	通用	函数，传入的参数格式为 {x,y,width,height}
onScroll	当文本输入组件为多行模式时，发生滚动行为会调用此回调	通用	函数，会将滚动位置以如下参数格式传入：{ nativeEvent: { contentOffset: { x, y } } }
onSelectionChange	当选中的文字改变时调用的回调	通用	函数，参数会将选中的字符范围以如下格式传入：{ nativeEvent: { selection: { start, end } } }
onSubmitEditing	当用户按下回车键时调用的回调（如果是多行模式，则此属性无效）	通用	函数

（续表）

属性名	解释	平台	值类型或可选值
placeholder	如果没有任何文字输入，则会显示这个属性的值	通用	字符串
placeholderTextColor	默认显示文字的文字颜色	通用	特定格式的颜色字符串
returnKeyType	回车键显示的文案	通用	枚举字符串： 通用： • done • go • next • search • send 安卓： • none • previous 苹果： • default • emergency-call • google • join • route • yahoo
secureTextEntry	设置是否密文输入	通用	布尔值 • true：密文输入 • false：明文输入
selectTextOnFocus	设置是否获取焦点的时候选中所有文字	通用	布尔值
selection	设置选中文字	通用	如下格式的对象： {start:number,end:number}
selectionColor	设置文字选中区域的颜色	通用	特定格式的颜色字符串
value	文本框中的文字内容	通用	这个属性如果设置，组件中的文本将一直与这个值保持一致
inlineImageLeft	设置组件左侧图片	Android	本地图片引用
underlineColorAndroid	设置文本组件的下画线颜色	Android	特定格式的颜色字符串
numberOfLines	设置文本输入组件的行数	Android	数值
inlineImagePadding	设置图片与组件间的距离	Android	数值
clearButtonMode	设置文本输入组件右侧是否显示“清除”按钮	iOS	枚举字符串： • never：永不显示 • while-editing：编辑时显示 • unless-editing：非编辑时显示 • always：始终显示

（续表）

属性名	解释	平台	值类型或可选值
clearTextOnFocus	设置每次开始输入时是否清除内容	iOS	布尔值
enablesReturnKeyAutomatically	设置当组件中没有文字时是否禁用确认按钮	iOS	布尔值，默认为 false
dataDetectorTypes	设置文本中链接的模式	iOS	枚举字符串： • phoneNumber：电话 • link：链接 • address：地址 • calendarEvent：日历事件 • none：无 • all：全部
keyboardAppearance	设置键盘的颜色	iOS	枚举字符串： • default：默认 • light：亮色 • dark：暗色
onKeyPress	当键盘中的键被按下时调用的回调	iOS	函数，会在 onChange 之前调用，传入的参数格式为： { nativeEvent: { key: keyValue } }

TextInput 组件也提供了两个实例方法，用来操作组件本身，定义如表 6-13 所示。

表 6-13　TextInput 组件的两个实例方法

方法名	解释	平台	参数定义
isFocused	获取当前文本输入组件是否获取了焦点	通用	无
clear	清空当前文本输入组件的内容	通用	无

实例方法应用实例如下：

```
return(
            <TextInput style={baseStyle}
            autoCapitalize={'words'}
            autoCorrect={false}
            autoFocus={true}
            caretHidden={true}
            defaultValue="HelloWorld"
            multiline={false}
            onBlur={()=>{
                console.log('onBlur');
            }}
            onChangeText={(text)=>{
                console.log(text);
            }}
```

```
                onEndEditing={()=>{
                    console.log(this.refs.textinput.isFocused());
                    this.refs.textinput.clear();
                }}
                onFocus={()=>{
                    console.log("focus");
                }}
                onScroll={({ nativeEvent: { contentOffset: { x, y } } })=>{
                    console.log(x,y);
                }}
                selection={{start:0,end:5}}
                selectionColor='red'
                keyboardAppearance={'dark'} ref="textinput"/>
            );
```

抽 丝 剥 茧

上面示例代码中使用到了 ref 这个属性，这个属性可以为组件设置标识，其承载类对象可以通过 refs 属性获取到对应的组件实例。这种获取组件实例的方法在开发中将大量使用。

6.8　StatusBar 状态栏组件的应用

无论是 iOS 还是 Android 平台，在屏幕的最上方都默认显示一个状态栏。所谓状态栏，其实就是显示时间、网络、电量等设备信息的地方。在 React Native 中，StatusBar 组件用来操作状态栏。需要注意的是，StatusBar 组件可以在多个地方使用，但状态栏永远只有一个，后设置的 StatusBar 会将先设置的覆盖。

打开 HelloWorld 工程，在 Demo 文件夹下新建一个命名为 StatusBarDemo.js 的文件，在其中编写如下测试代码：

```
    import React,{Component} from 'react';
    import {StatusBar,Button,View} from 'react-native';
    export default class StatusBarDemo extends Component{
        constructor(props){
            super(props);
            this.state = {
                hidden:false
            }
        }
        render(){
            return(
                <View>
                <StatusBar ref="status"
```

```
                hidden={this.state.hidden}
                animated={true}
                backgroundColor='red'
                translucent={true}
                barStyle={'dark-content'}
                showHideTransition={'slide'}/>
                <Button title="button"
                style={{top:100}}
                onPress={()=>{
                    this.setState({
                            hidden:!(this.refs.status.props.hidden)
                        });
                    console.log(StatusBar.currentHeight);
                }}/>
                </View>
            );
        }
    }
```

修改 index.ios.js 与 index.Android.js 文件如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
} from 'react-native';
import StatusBarDemo from './Demo/StatusBarDemo';
export default class HelloWorld extends Component {
  render() {
    return (<StatusBarDemo />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

上面的示例代码实现了一个小行为，当用户单击按钮时，状态栏会切换显隐状态，并且有一个动画特效。

StatusBar 组件提供给了我们一个静态属性（见表 6-14），可以直接使用类名来进行调用。

表 6-14　StatusBar 组件的静态属性

静态属性	解释	平台	备注
currentHeight	获取状态栏的高度	Android	在 iOS 平台，状态栏高度都为 20

StatusBar 中提供的常用属性如表 6-15 所示。

表 6-15　StatusBar 的常用属性

属性名	解释	平台	值类型或可选值
animated	状态栏的变化是否以动画的形式呈现，包括显隐、背景色、风格的变化	通用	布尔值
hidden	设置是否隐藏状态栏	通用	布尔值
barStyle	设置状态栏风格	通用	枚举字符串 • default：默认 • light-content：内容为白色 • dark-content：内容为黑色
backgroundColor	设置状态栏背景色	Android	特定格式的颜色字符串
translucent	设置状态栏是否为沉浸式	Android	布尔值
networkActivityIndicatorVisible	设置是否显示网络提示符	iOS	布尔值
showHideTransition	设置显隐状态栏时的动画效果	iOS	枚举字符串 • fade：渐隐渐现 • slide：收入收出

6.9　Picker 选择器组件的应用

Picker 组件是 React Native 中一个跨平台的选择器控件，对于某些例如城市选择、日期选择、性别选择等需求，使用这个组件美观而实用。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 PickerDemo.js 的文件，在其中编写如下测试代码：

```
import React,{Component} from 'react';
import {Picker} from 'react-native';
export default class PickerDemo extends Component{
    constructor(props){
        super(props);
        this.state = {
            value:1
        }
    }
    render(){
        return(
            <Picker style={{top:100,marginLeft:30,marginRight:30,
height:100}}
            onValueChange={(itemValue,itemPosition)=>{
                console.log(itemValue,itemPosition);
                this.setState({
                    value:itemPosition
```

```
                        });
                }}
                selectedValue={this.state.value}
                mode={'dialog'}
                itemStyle={{fontSize:24}} ref='picker'>
                    <Picker.Item label="iOS" value={0} color='red'/>
                    <Picker.Item label="Android" value={1} color='blue'/>
                    <Picker.Item label="Web" value={2} color='green'/>
                </Picker>
            );
        }
    }
```

注意，Picker 组件是一个受控组件，即其表现的状态只与 selectedValue 属性有关，用户的操作并不能修改 Picker 组件的值。

将 index.ios.js 与 index.Android.js 文件修改如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
} from 'react-native';
import PickerDemo from './Demo/PickerDemo';
export default class HelloWorld extends Component {
  render() {
    return (<PickerDemo />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

运行工程，Picker 组件在 iOS 平台与 Android 平台的呈现分别如图 6-20 与图 6-21 所示。





图 6-20　iOS 平台的 Picker 组件

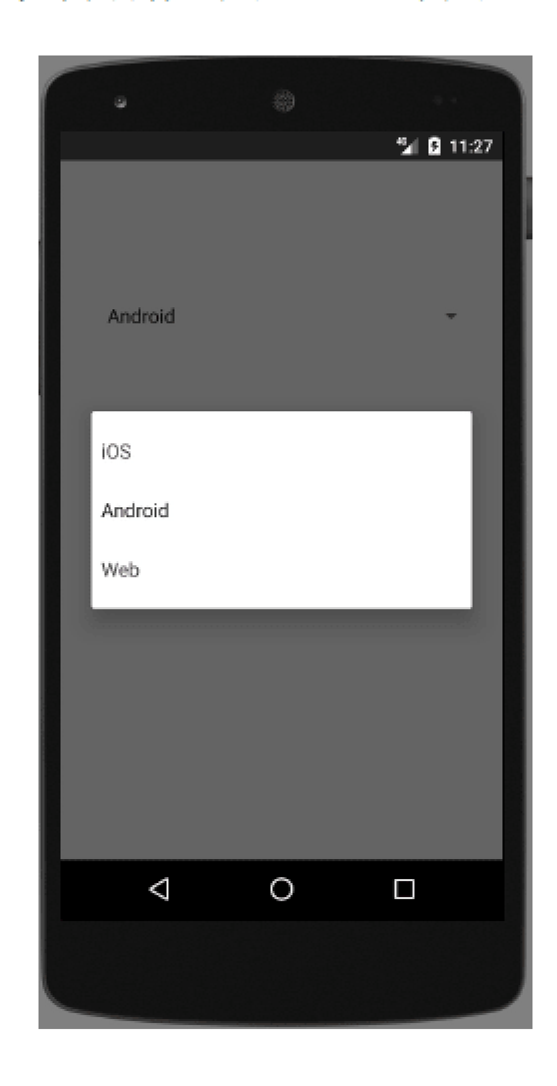


图 6-21　Android 平台的 Picker 组件

Picker 组件在使用时需要使用双标签，其内部组合 PickerItem 来表示选择器中的每一行数据。Picker 组件可以设置的属性如表 6-16 所示。

表 6-16　Picker 组件可以设置的属性

属性名	解释	平台	值类型或可选值
selectedValue	选中的值	通用	字符串或数值
style	设置布局样式	通用	样式对象
onValueChange	当用户操作值改变时调用的回调	通用	函数，参数如下： itemValue：item 对应的值 itemPosition：item 对应的位置
enabled	是否可以进行用户交互	Android	布尔值
mode	设置选择器样式	Android	枚举字符串 dialog：对话框演示 dropdown：下拉框样式
prompt	设置提示字符串	Android	字符串
itemStyle	指定 Item 的布局样式	iOS	样式对象

实际上，Picker.item 也是一个独立的类，用来创建选择器组件中每条具体数据的呈现载体，其中可进行设置的属性如表 6-17 所示。

表 6-17　Picker.item 组件可以设置的属性

属性名	解释	平台	值类型或可选值
label	设置 Item 显示的字符串	通用	字符串
value	设置 Item 对应的值	通用	字符串或数值
color	设置 Item 的文字颜色	通用	特定的颜色字符串

6.10　Modal 模态视图组件的应用

Modal 模态视图是 React Native 中一个十分强大的跨平台容器视图，你可以使用它任意组合其他组件来创建自定义的对话框。在 HelloWorld 工程的 Demo 文件夹下面创建一个命名为 ModalDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {Modal,Text,View,Button} from 'react-native';
export default class ModalDemo extends Component{
    constructor(props){
        super(props);
        this.state = {
            hidden:true
        }
    }
```

```
        render(){
            return(
                <View>
                <Modal visible={!this.state.hidden}
                transparent={false}
                animationType={'slide'}
                onRequestClose={()=>{
                    console.log('close');
                }}
                onShow={()=>{
                    console.log('onShow');
                }}
        supportedOrientations={['portrait','portrait-upside-down','landscape',
'landscape-left', 'landscape-right']}
                onOrientationChange={()=>{
                    console.log("onOrientationChange");
                }}
                hardwareAccelerated={true}
                >
                    <Text style={{top:150,textAlign:'center'}}>I'm a
Modal!</Text>
                    <View style={{top:180}}>
                        <Button title="Close"
                        onPress={()=>{
                            this.setState({
                                hidden:true
                            });
                        }}/>
                    </View>
                </Modal>
                <View style={{top:30}}>
                    <Text style={{textAlign:'center',fontSize:22}}>Hello
Modal!</Text>
                    <Button title="Show Modal"
                    onPress={()=>{
                        this.setState({
                            hidden:false
                        });
                    }}/>
                </View>
                </View>
            );
        }
    }
```

注意，Modal 视图会覆盖在当前界面的最上层，上面的示例代码中，onRequestClose 属性比较特殊，当接收到 Modal 视图的关闭请求时会被调用（并非隐藏）。在 Android 平台上这个属性是必须提供的，当用户单击 Android 设备的返回键时，这个属性设置的回调会被调用。Modal 模态视图常用属性如表 6-18 所示。

表 6-18　Modal 模态视图常用属性

属性名	解释	平台	值类型或可选值
animationType	设置模态视图显示时的动画效果	通用	枚举字符串 slide：切入切出 fade：渐隐渐现
onRequestClose	当接收到关闭请求时调用的回调	通用	函数，在 Android 平台为必需
onShow	当模态视图显示时调用的回调	通用	函数
transparent	设置模态视图是否覆盖背景	通用	布尔值 true：模态视图背景透明 false：模态视图背景不透明
visible	设置模态视图是否可见	通用	布尔值
hardwareAccelerated	设置是否硬件加速	Android	布尔值
supportedOrientations	设置支持的屏幕方向	iOS	枚举字符串数组，下面是可选择的值 'portrait', 'portrait-upside-down', 'landscape', 'landscape-left', 'landscape-right'
onOrientationChange	模态框显示时屏幕方向发生改变调用的回调	iOS	函数，只有当水平和竖直方向切换时会调用

注意，在 iOS 平台中，应用是否支持横竖屏的切换是需要在工程的 Info.plist 文件中进行配置的，使用 Xcode 打开 iOS 工程，在其中的 Info.plist 文件中添加如图 6-22 中所示的键值。（图 6-22 中设置了支持 Home 键在下、Home 键在左和 Home 键在右 3 种设备方向。）

▼ Supported interface orientations	Array	(3 items)
Item 0	String	Portrait (bottom home button)
Item 1	String	Landscape (left home button)
Item 2	String	Landscape (right home button)

图 6-22　在 Info.plist 文件中配置设备支持的方向

6.11　KeyboardAvoidingView 组件的应用

React Native 中的 KeyboardAvoidingView 并不是一个十分完善的组件，其设计的初衷是为了解决在移动设备上当有键盘弹出时视图可以自适应地调整位置。KeyboardAvoidingView 是一个容器组件，其中放入的视图组件在键盘弹出时会根据键盘的高度进行自适应的位置调整。在 HelloWorld 工程中的 Demo 文件夹下新建一个 KeyboardAvoidingViewDemo.js 的文件，编写如下代码：

```
    import React,{Component} from 'react';
    import {KeyboardAvoidingView,TextInput,View,Button} from 'react-native';
    export default class KeyboardAvoidingViewDemo extends Component{
        render(){
            return(
                <View style={{justifyContent:'flex-end',bottom:0,flex:1}}>
                <Button title='TEXT' onPress={()=>{
                    console.log("===");
                }}/>
                <KeyboardAvoidingView behavior={'position'}>
                    <TextInput style={{borderWidth:1,borderColor:'gray',
height:30}}/>
                </KeyboardAvoidingView>
                </View>
            );
        }
    }
```

上面的示例代码创建了一个 Button 组件和一个 KeyboardAvoidingView 组件，修改 index.ios.js 与 index.Android.js 文件后在 iOS 平台运行项目。默认界面与键盘弹出时的界面如图 6-23 与图 6-24 所示。

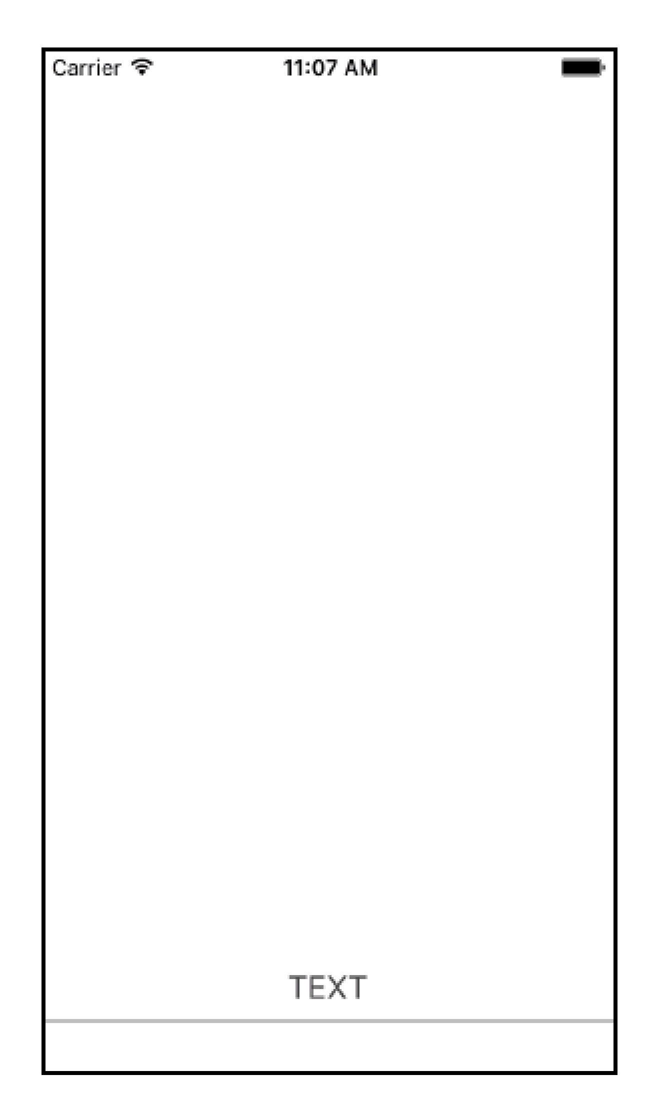




图 6-23　默认状态下的输入框

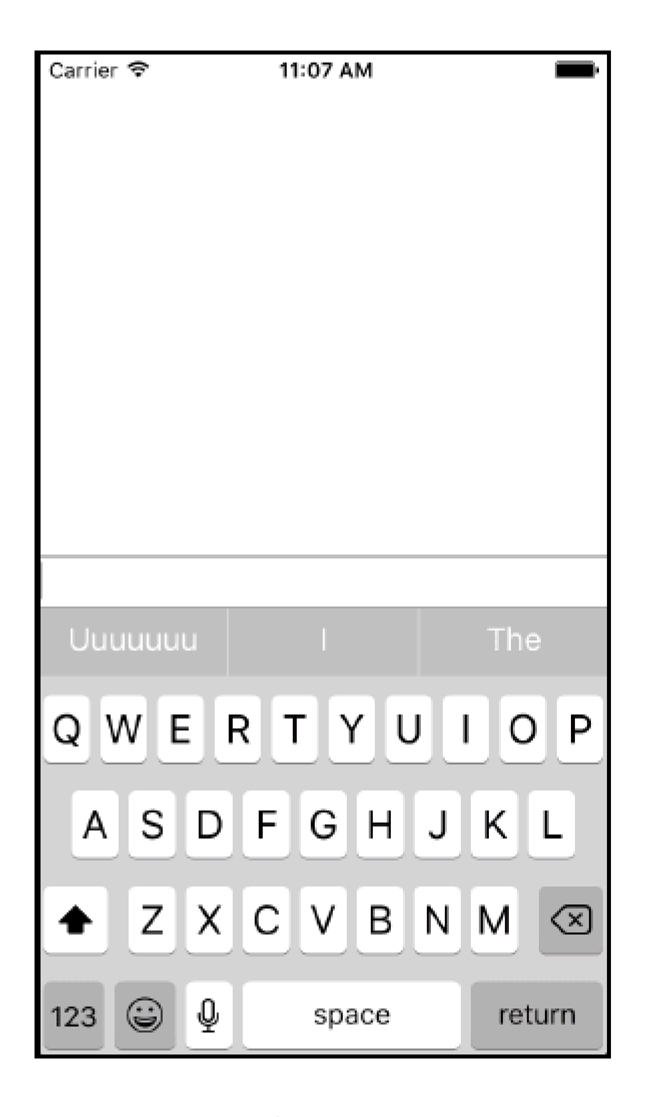


图 6-24　键盘弹出时的输入框

你可能已经发现了，输入框跟随键盘的弹出进行了移动，但是按钮组件并没有移动。我们可以通过设置 KeyboardAvoidingView 组件的 behavior 属性为 padding 来实现所有输入框组件上面的组件一起移动的效果。

KeyboardAvoidingView 组件支持的属性见如表 6-19 所示。

表 6-19　KeyboardAvoidingView 组件支持的属性

属性名	解释	平台	值类型或可选值
behavior	定义行为	通用	枚举字符串 • height：高度模式，这种模式下，组件不随键盘做适应 • position：这种模式下，组件会随着键盘的状态改变位置 • padding：这种模式下，组件会随着键盘的状态调整自己的 padding
contentContainerStyle	当组件的行为设置为 position 模式时，会自动创建一个内容容器视图，这个属性设置内容视图容器视图的 style	通用	style 风格对象
keyboardVerticalOffset	这个属性用于距离调整的补偿，例如，组件可能本来就距离界面下方有一定距离，键盘弹出时默认会将组件的位置修改为一个键盘高度的距离，设置这个值可以补偿距离	通用	数值

注意，keyboardVerticalOffset 属性非常有用，很多情况下，我们的输入组件并不一定是紧贴界面底部的，通过这个补偿距离可以完美地控制组件的位置适应，示例如下：

```
        render(){
            return(
                <View style={{justifyContent:'flex-end',bottom:10,flex:1}}>
                <Button title='TEXT' onPress={()=>{
                    console.log("===");
                }}/>
                <KeyboardAvoidingView behavior={'padding'}
keyboardVerticalOffset={10}
                >
                    <TextInput style={{borderWidth:1,borderColor:'gray',
height:30}}/>
                </KeyboardAvoidingView>
                </View>
            );
        }
```

抽 丝 剥 茧

在本节的开始就提到，KeyboardAvoidingView 组件并不是一个非常完善的组件，在 iOS 平台其表现很好，但在 Android 平台则因系统版本的不同造成很大差异，后面将会介绍如何在项目中根据不同平台来适配代码。继续学习，你一定会开发出极佳体验的跨平台移动应用。

6.12 WebView 网页组件的应用

无论是在 iOS 平台还是 Android 平台，WebView 都是一种十分重要的网页视图控件。有了 WebView 的支持，应用程序的内容会更加丰富灵活。在 React Native 中提供了跨平台的 WebView 组件来进行网页视图的渲染。

6.12.1 WebView 常用属性解析

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 WebViewDemo.js 的文件。在其中编写如下代码：

```
import React,{Component} from 'react';
import {WebView} from 'react-native';
export default class WebViewDemo extends Component{
    render(){
        return(
            <WebView source={{uri:'https://www.baidu.com'}}
             bounces={false}
             dataDetectorTypes={['link']}
             decelerationRate={'normal'}
             domStorageEnabled={true}
             injectedJavaScript="var a = 100;"
             scalesPageToFit={true}/>
             );
    }
}
```

上面的代码在界面中创建了一个 WebView 视图，默认加载百度网站的首页，修改 index.ios.js 文件与 index.Android.js 文件后运行工程，效果如图 6-25 所示。



图 6-25 WebView 组件效果

抽 丝 剥 茧

如果加载的网页是 HTTP 协议的，在 iOS 平台需要设置工程的 Info.plist 文件使其支持，关于这部分内容在 Image 组件一节有介绍，如果忘记了，可以翻回去查看。

WebView 组件的 soure 属性除了可以加载一个网址外，还可以加载本地的 HTML 文件或者 HTML 字符串，找一个网页保存为 HTML 文件，将其放入 React Native 工程的 source 文件夹下，修改 WebViewDemo 类的 render 方法如下：

```
render(){
    return(
        <WebView source={require('./../source/test.html')}
         bounces={false}
         dataDetectorTypes={['link']}
         decelerationRate={'normal'}
         domStorageEnabled={true}
         injectedJavaScript="var a = 100;"
         scalesPageToFit={true}/>
         );
}
```

运行工程，会看到 HTML 文件中的内容被解析成了 WebView 网页。使用 HTML 字符串加载网页视图的逻辑也一样，这里就不再演示了，在表 6-20 的属性列表中有详细的用法介绍。

表 6-20　WebView 的属性

属性名	解释	平台	值类型或可选值
injectedJavaScript	在加载网页之前注入一段 JavaScript 代码	通用	字符串
mediaPlaybackRequiresUserAction	设置网页中的视频是否接收到用户交互后再进行播放	通用	布尔值
scalesPageToFit	设置是否把网页缩放到合适比例	通用	布尔值
source	设置网页资源	通用	可以有 3 种设置方式 （1）设置为网页地址，格式如下： {uri: string, method: string, headers: object, body: string} 其中，method 设置请求方式，可选“GET”与“POST”，headers 设置请求头内容，body 设置请求体。 （2）使用 require 方法引入本地 HTML 资源 （3）{html: string, baseUrl: string}

（续表）

属性名	解释	平台	值类型或可选值
startInLoadingState	是否强制在加载时显示 loading 组件	通用	布尔值
style	UI 风格	通用	和 View 组件一致
allowsInlineMediaPlayback	设置视频在网页中直接播放或跳转原生播放器	iOS	布尔值
bounces	设置网页视图是否有回弹效果	iOS	布尔值
dataDetectorTypes	设置探测网页中的特殊符号类型	iOS	这个属性可以将电话、地址、日历等信息进行探测，添加超链接。可以设置为数组对象，其中可选枚举如下： • phoneNumber：电话号 • link：链接 • address：地址 • calendarEvent：日历事件 • none：无探测 • all：全部探测
decelerationRate	设置滑动速度	iOS	枚举字符串 • normal：正常速度 • fast：快速
scrollEnabled	是否允许滑动	iOS	布尔值
domStorageEnabled	是否开启 DOM 本地存储	Android	布尔值
JavaScriptEnabled	是否开启 JavaScript	Android	布尔值
userAgent	设置 WebView 的 user-agent 字段	Android	字符串

除了上面提到的属性外，WebView 组件中还有两个十分强大的属性，分别用来设置网页视图加载时的等待视图和网页加载出错的提示视图，如表 6-21 所示。

表 6-21　WebView 组件的属性

属性名	解释	平台	值类型或可选值
renderError	设置一个函数，用于显示加载网页出错时的提示视图	通用	函数，需要返回一个组件
renderLoading	设置一个函数，用于显示网页加载过程中的等待视图	通用	函数，需要返回一个组件，默认将返回一个居中的 ActivityIndicatorView 组件

示例代码如下：

```
render(){
    return(
        <WebView source={{uri:'https://www.baidu.com'}}
         bounces={false}
```

```
            dataDetectorTypes={['link']}
            decelerationRate={'normal'}
            domStorageEnabled={true}
            injectedJavaScript="var a = 100;"
            scalesPageToFit={true}
            startInLoadingState={true}
            renderLoading={()=>{
                return(
                    <Text>Loading</Text>
                );
             }}
             renderError={()=>{
                 return(
                     <Text>Load Error</Text>
                 );
             }}/>
        );
    }
```

6.12.2　WebView 加载过程监听相关属性

监听 WebView 的加载过程十分有必要，有时候我们甚至需要拦截某些请求来进行定制化的逻辑处理。表 6-22 所示的 WebView 组件属性用来监听网页的加载过程。

表 6-22　WebView 组件中用来监听网页加载过程的属性

属性名	解释	平台	值类型或可选值
onError	网页加载出错时调用的回调	通用	函数
onLoadStart	网页开始加载时调用的回调	通用	函数
onLoad	网页加载成功时调用的回调	通用	函数
onLoadEnd	网页加载完成时调用的回调，无论是否成功	通用	函数
onNavigationStateChange	网页导航发生变化时调用的回调	通用	函数
onShouldStartLoadWithRequest	网页发起请求时调用的回调	iOS	函数，需要返回一个布尔值，如果返回 false 则不允许此次请求，返回 true 则表示允许

在 iOS 原生开发中，经常使用到 WebView，在原生与 WebView 之间经常会遇到数据互传的需求，一种解决方案是拦截 WebView 的请求 url 进行协议解析，这就用到 onShouldStartLoadWithRequest 属性所对应的回调了，这个属性对应的函数中会传入一个 Web 对象，可以通过获取其中的 url 来进行协议解析，示例如下：

```
    render(){
        return(
```

```
        <WebView source={{uri:'https://www.baidu.com'}}
            onError={()=>{
            console.log("error");
            }}
            onLoad={()=>{
               console.log("load");
            }}
            onLoadEnd={()=>{
                console.log("load end");
            }}
            onLoadStart={()=>{
                console.log("load start");
            }}
            onNavigationStateChange={(obj)=>{
                console.log("navigation",obj);
            }}
            onShouldStartLoadWithRequest={(request)=>{
                console.log("request",request.url);
                return true;
            }}/>
        );
    }
```

使用协议解析的这种方式进行 React Native 与 JavaScript 交互，其参数通常是放入 url 中的，这种方式只能完成一些简单的交互任务，对于更复杂的传参需求，将通过下一小节的方法来实现。

6.12.3 React Native 与 WebView 交互

React Native 与 WebView 交互还可以通过 WebView 组件中的 onMessage 属性来完成。这个属性需要设置为一个函数对象，在 WebView 中使用 window.postMessage 方法会调用此属性对应的函数，从而实现 React Native 与 JavaScript 的数据交互。网页端在调用 postMessage 方法时，只能传入一个字符串类型的参数，示例代码如下：

```
    render(){
        return(
        <WebView source={{uri:'https://www.baidu.com'}}
         onMessage={({nativeEvent:{data}})=>{
         console.log(data);
        }}/>
        );
    }
```

需要注意，网页端传过来的参数会封装进 nativeEvent 属性的 data 属性中，可以使用如上代码所示的解构赋值方法直接抓取到 data 数据。要测试 React Native 与 WebView 的交互我们需要借助 Safari 浏览器的开发者工具。在 iOS 模拟器上运行工程，在 Chrome 浏览器中打开 React Native 的

调试模式，打开 Safari 浏览器，选择菜单栏中的“开发→Simulator”选项，可以看到在模拟器上运行的 JavaScript 线程，其中 RCTJSContext 是 React Native 中的 JavaScript 线程，www.baidu.com 为 WebView 中的 JavaScript 线程，选择 www.baidu.com，如图 6-26 所示。



图 6-26　使用 Safari 开发者工具调试 WebView

在弹出的调试台界面中输入如下代码：

```
window.postMessage("Hello World");
```

效果如图 6-27 所示。

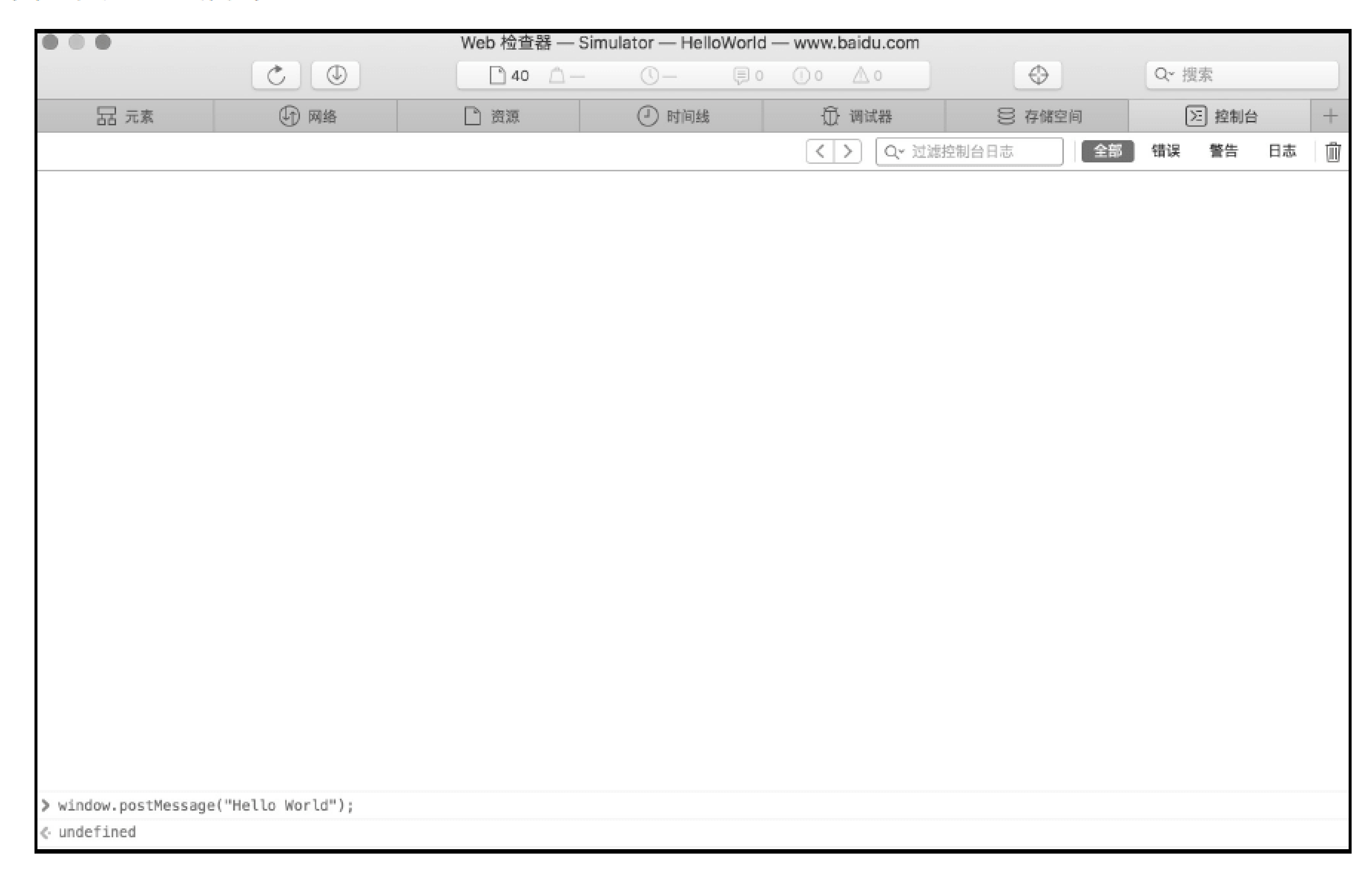


图 6-27　在 WebView 中执行 JavaScript 脚本

回到 Chrome 浏览器中 React Native 应用调试界面，可以看到在工作台输出了 WebView 传递进来的数据 Hello Wrold。

6.13　View 视图组件的应用

通过前面的章节，你学习到了许多 React Native 中的独立视图组件，它们中很多都是基于 View 组件扩展而来的。本节将向你介绍 React Native 中视图组件的根基：View 组件。

6.13.1 View 组件 Style 属性的解析

作为界面中最基础的组件，View 的 Style 样式支持的属性非常丰富，大致可以分为 4 类：布局类、阴影类、几何变换类和样式类。

React Native 中的界面布局实际上采用的是 FlexBox 模型，关于 FlexBox 布局后面会有专门章节进行讲解，本节不做过多叙述。阴影类相关的属性在 Image 组件一节有完整的介绍，其只支持 iOS 平台。几何变换的属性在 Image 组件一节也有详细的介绍，即 Transfrom 相关属性，用来对视图进行平移、旋转、缩放等。如果你对它们的用法有些模糊，可以返回相关章节进行回顾。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ViewDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {View} from 'react-native';
export default class ViewDemo extends Component{
    render(){
        return(
            <View style={baseStyle}>
            </View>
        );
    }
}
let baseStyle = {
    //布局属性
    width:200,
    height:200,
    //样式属性
    backfaceVisibility:'hidden',
    backgroundColor:'red',
    borderTopColor:'green',
    borderRightColor:'blue',
    borderBottomColor:'black',
    borderLeftColor:'yellow',
    borderTopWidth:10,
    borderBottomWidth:5,
    borderLeftWidth:20,
    borderRightWidth:2,
    borderStyle:'solid',
    opacity:0.9,
    overflow:'visible'
}
```

修改 index.ios.js 与 index.Android.js 文件后，运行工程，效果如图 6-28 所示。

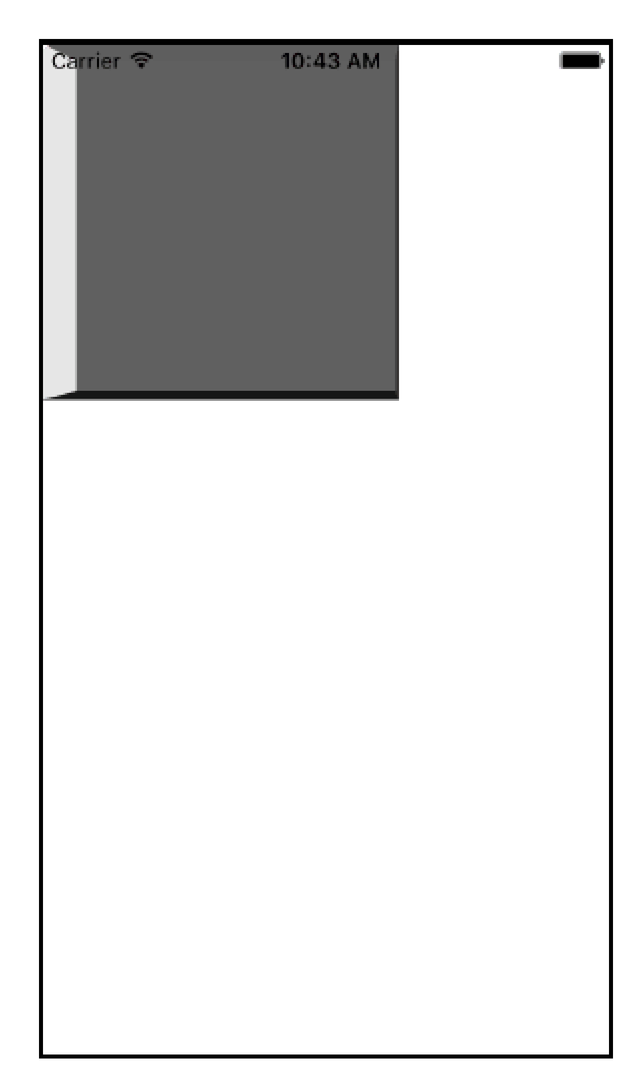


图 6-28　View 组件效果

View 组件的 Style 风格常用属性如表 6-23 所示。

表 6-23　View 组件的 Style 风格常用属性

属性名	解释	平台	值类型或可选值
backfaceVisibility	设置视图的背面是否可见	通用	枚举字符串 • visible：可见 • hidden：不可见
backgroundColor	设置视图背景色	通用	特定的颜色字符串
borderColor	设置边框颜色	通用	特定的颜色字符串
borderTopColor	上边框颜色	通用	特定的颜色字符串
borderRightColor	右边框颜色	通用	特定的颜色字符串
borderBottomColor	下边框颜色	通用	特定的颜色字符串
borderLeftColor	左边框颜色	通用	特定的颜色字符串
borderRadius	设置边框圆角半径	通用	数值
borderTopLeftRadius	设置边框左上角圆角半径	通用	数值
borderTopRightRadius	设置边框右上角圆角半径	通用	数值
borderBottomLeftRadius	设置边框左下角圆角半径	通用	数值
borderBottomRightRadius	设置边框右下角圆角半径	通用	数值
borderStyle	设置边框风格	通用	枚举字符串 • soild：实线 • dotted：点线 • dashed：虚线
borderTopWidth	上边框宽度	通用	数值
borderRightWidth	右边框宽度	通用	数值

（续表）

属性名	解释	平台	值类型或可选值
borderBottomWidth	下边框宽度	通用	数值
borderLeftWidth	左边框宽度	通用	数值
opacity	设置透明度	通用	数值 0～1
overflow	设置溢出是否隐藏	通用	枚举字符串 • visible：可见 • hidden：隐藏

抽 丝 剥 茧

View 组件的风格属性在 iOS 平台都有很好的支持，在 Android 平台可能会有些差异。例如，同时设置了圆角和边框颜色，边框颜色的属性可能无法正常渲染。

6.13.2 View 组件基础属性的解析

View 组件中提供了许多基础属性，如表 6-24 所示。

表 6-24 View 组件基础属性

属性名	解释	平台	值类型或可选值
accessibilityLabel	设置读屏软件读到此组件时的值	通用	字符串
accessible	设置是否开启读屏功能	通用	布尔值
onLayout	当组件挂载完成或者布局变化时调用的回调	通用	函数，传入参数格式如下：{nativeEvent: { layout: {x, y, width, height}}}
onMoveShouldSetResponder	当触摸点在组件上移动时调用的方法，这个方法需要返回一个布尔值决定是否将组件设置为事件的响应者	通用	函数，需要返回布尔值
onMoveShouldSetResponderCapture	当触摸点在组件上移动时，设置组件是否拦截此事件（若拦截，则事件不会传递到子视图）	通用	函数，需要返回布尔值
onResponderGrant	当组件成功成为事件响应者时被调用	通用	函数
onResponderMove	当作为事件响应者接收到移动事件时调用	通用	函数
onResponderReject	当成为事件响应者失败时被调用	通用	函数
onResponderRelease	触摸事件处理完成后的回调，注销响应	通用	函数

（续表）

属性名	解释	平台	值类型或可选值
onResponderTerminate	这个属性设置的函数在用户触摸交互被打断时调用，例如来电话	通用	函数
onResponderTerminationRequest	当接收到中断事件请求时被调用	通用	函数，需要返回布尔值表示是否接受此中断
onStartShouldSetResponder	当触摸事件开始的时候回调此函数，需要返回布尔值决定是否成为响应者	通用	函数，需要返回布尔值
onStartShouldSetResponderCapture	当触摸事件开始时回调的函数，返回布尔值确定是否拦截事件	通用	函数，需要返回布尔值
pointerEvents	设置视图是否可以作为触摸事件的目标	通用	枚举字符串 • auto：可以作为触摸事件的目标 • none：不能作为触摸事件的目标 • box-none：视图自身不能成为触摸事件的目标，但是子视图可以作为触摸事件的目标 • box-only：视图本身可以作为触摸事件的目标，但是子视图不能成为触摸事件的目标
removeClippedSubviews	对于滑动组件，其子视图很可能会超出可见范围，这个属性设置超出可见范围的子视图是否被自动移除	通用	布尔值

需要注意，上面提出的 View 组件的属性虽然很多与用户触摸事件有关，但一般情况下你不会直接使用到 View 组件的交互功能，如果需要自定义用户交互组件，可以使用 Touchable 相关组件来制作。

6.14　Touchable 相关交互组件的应用

在前面章节中，你学习到了使用 Button 组件来创建按钮，然而 Button 组件的定制化性并不强，例如想要创建一个同时由图标和文字两部分组成的按钮，使用 Button 组件就显得有些棘手。本节我将向你介绍 React Native 中与用户触摸交互功能相关的组件，使用这些组件，你可以轻松地创建出自定义的按钮。

6.14.1 TouchableWithoutFeedback

React Native 中 Touchable 相关的组件有 TouchableWithoutFeedback 组件、TouchableOpacity 组件、TouchableNativeFeedback 组件和 TouchableHigglight 组件。上面提到的组件中后几个组件都是基于第一个组件而扩展出来的。

TouchableWithoutFeedback 组件用来响应用户的触摸操作，需要注意的是，这个组件中只能包含一个子组件，如果要创建复合的按钮控件，需要使用 View 组件进行包装。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 TouchableWithoutFeedbackDemo.js 的文件。在其中编写如下代码：

```
import React,{Component} from 'react';
import {View,TouchableWithoutFeedback,Text,Image} from 'react-native';
export default class TouchableWithoutFeedbackDemo extends Component{
   render(){
      return(
         <View style={{top:100,left:100}}>
            <TouchableWithoutFeedback
             onLongPress={()=>{
                console.log("long press");
             }}
             onPress={()=>{
                console.log("press");
             }}
             onPressIn={()=>{
                console.log("press in");
             }}
             onPressOut={()=>{
                console.log("press out");
             }}>
             <View>
                <Image source={require('./../source/image.png')}
                 style={{width:50,height:50}}/>
                <Text>Button</Text>
             </View>
           </TouchableWithoutFeedback>
        </View>
     );
   }
}
```

修改 index.ios.js 文件与 index.Android.js 文件后运行工程，打开调试模式，对自定义的交互组件进行单击、长按等手势可以看到调用了相应的触发方法。

需要注意，TouchableWithoutFeedback 组件是无 UI 上用户反馈的交互组件，一般情况下你不会真正使用到这个组件，在做移动端应用设计时，良好的反馈是必要的。

TouchableWithoutFeedback 组件常用属性见表 6-25。

表 6-25　TouchableWithoutFeedback 组件常用属性

属性名	解释	平台	值类型或可选值
delayLongPress	长按触发事件	通用	数值 设置用户长按多久后触发长按回调
delayPressIn	按下触发事件	通用	数值 设置用户按下多久后触发按下回调，即 onPressIn 属性的回调
delayPressOut	抬起触发事件	通用	数值 设置用户抬起手指多久后触发抬起回调，即 onPressOut 属性的回调
disabled	设置是否禁用组件	通用	布尔值
hitSlop	设置交互组件可交互的范围	通用	如下格式对象： {top: number, left: number, bottom: number, right: number} 这个属性可以将按钮的可交互范围变大
onLayout	当组件挂载完成或布局改变时调用的回调	通用	函数，会传入如下格式的参数： {nativeEvent: {layout: {x, y, width, height}}}
onLongPress	设置长按触发函数	通用	函数
onPress	设置单击触发函数	通用	函数 手指按下并抬起时触发
onPressIn	设置按下触发函数	通用	函数 手指按下时触发
onPressOut	设置抬起触发函数	通用	函数 手指抬起时触发
pressRetentionOffser	这个属性设置当手指按下不抬起并在屏幕上移动时，移动多远距离后组件将不再激活	通用	如下格式对象： {top: number, left: number, bottom: number, right: number}

6.14.2　TouchableOpacity

了解了 TouchableWithoutFeedback 组件后，学习其他 Touchable 相关组件将容易很多。TouchableOpacity 组件的基本用法和 TouchableWithoutFeedback 组件一致，也就是说，所有 TouchableWithoutFeedback 组件可以使用的属性在 TouchableOpacity 组件中都可以使用，TouchableOpacity 组件在进行用户交互的时候会产生一个透明度变化的反馈。在 HelloWorld 工程的 Demo 文件夹下面新建一个命名为 TouchableOpacityDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {View,TouchableOpacity,Text,Image} from 'react-native';
export default class TouchableOpacityDemo extends Component{
    render(){
```

```
        return(
            <View style={{top:100,left:100}}>
                <TouchableOpacity
                 onLongPress={()=>{
                     console.log("long press");
                 }}
                 onPress={()=>{
                     console.log("press");
                 }}
                 onPressIn={()=>{
                     console.log("press in");
                 }}
                 onPressOut={()=>{
                     console.log("press out");
                 }}>
                    <View>
            <Image source={require('./../source/image.png')}
                      style={{width:50,height:50}}/>
              <Text>Button</Text>
         </View>
                   </TouchableOpacity>
              </View>
         );
    }
}
```

修改 index.ios.js 与 index.Android.js 文件后，运行工程，单击自定义的按钮组件来感受一下交互反馈的效果。

TouchableOpacity 组件除了支持所有 TouchableWithoutFeedback 组件中的属性外，还支持表 6-26 中的属性。

表 6-26　TouchableOpacity 组件支持属生

属性名	解释	平台	值类型或可选值
activeOpacity	设置反馈的透明度	通用	数值（0~1）

6.14.3　TouchableNativeFeedback

TouchableNativeFeedback 组件同样扩展于 TouchableWithoutFeedback 组件，其会带给用户一个原生体验的反馈效果。需要注意，TouchableNativeFeedback 组件只能应用在 Android 平台。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 TouchableNativeFeedbackDemo.js 的文件，编写如下代码：

```
import React,{Component} from 'react';
import {View,TouchableNativeFeedback,Text,Image} from 'react-native';
```

```
export default class TouchableNativeFeedbackDemo extends Component{
    render(){
        return(
            <View style={{top:100,left:100}}>
                <TouchableNativeFeedback
                 onLongPress={()=>{
                     console.log("long press");
                 }}
                 onPress={()=>{
                     console.log("press");
                 }}
                 onPressIn={()=>{
                     console.log("press in");
                 }}
                 onPressOut={()=>{
                     console.log("press out");
                 }}
                 background={TouchableNativeFeedback.Ripple('red', false)}>
                     <View>
                         <Image source={require('./../source/image.png')}
                          style={{width:50,height:50}}/>
                         <Text>Button</Text>
                     </View>
                </TouchableNativeFeedback>
            </View>
        );
    }
}
```

修改 index.Android.js 文件后在 Android 模拟器上运行工程，TouchableNativeFeedback 组件提供了 3 种原生体验的反馈效果，通过 background 属性（见表 6-27）来进行配置。

表 6-27　background 属性

属性名	解释	平台	值类型或可选值
background	决定原生体验类型	Android	有如下 3 种： • TouchableNativeFeedback.SelectableBackground()：选中效果的反馈 • TouchableNativeFeedback.SelectableBackgroundBorderless()：波纹效果的反馈 • TouchableNativeFeedback.Ripple(color, borderless)：波纹效果的反馈，开发者可以设置波纹颜色与是否扩展到组件边界之外，color 参数设置波纹的颜色，borderless 参数为布尔值，设置是否扩展到组件边界之外

6.14.4 TouchableHighlight

TouchableHighlight 组件当用户进行交互的时候会产生一个高亮的效果，其实现原理是降低组件的透明度并提供一个背景视图。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 TouchableHighlightDemo.js 的文件，编写如下代码：

```
import React,{Component} from 'react';
import {View,TouchableHighlight,Text,Image} from 'react-native';
export default class TouchableHighlightDemo extends Component{
    render(){
        return(
            <View style={{top:100,left:100}}>
                <TouchableHighlight
                 onLongPress={()=>{
                     console.log("long press");
                 }}
                 onPress={()=>{
                     console.log("press");
                 }}
                 onPressIn={()=>{
                     console.log("press in");
                 }}
                 onPressOut={()=>{
                     console.log("press out");
                 }}>
                <View>
                    <Image source={require('./../source/image.png')}
                     style={{width:50,height:50}}/>
                    <Text>Button</Text>
                </View>
               </TouchableHighlight>
          </View>
        );
    }
}
```

TouchableHighlight 组件的高亮背景也可以进行自定义，属性如表 6-28 所示。

表 6-28　TouchableHighlight 组件背景自定义属性

属性名	解释	平台	值类型或可选值
activeOpacity	设置组件激活时的透明度	通用	数值（0~1）
onHideUnderlay	当背景视图隐藏时调用的回调	通用	函数
onShowUnderlay	当背景视图显示时调用的回调	通用	函数

（续表）

属性名	解释	平台	值类型或可选值
style	风格属性	通用	同 View 组件 style 对象
underlayColor	设置背景视图的颜色	通用	特定格式的颜色字符串

到此，你已经掌握了 Touchable 相关组件的所有特性，可以开始随心所欲地构建自己的按钮了，发挥想象，多试几次吧！

6.15　ScrollView 滚动视图组件的应用

在移动应用开发中，列表的应用十分广泛。由于移动设备界面尺寸有限，因此在设计列表时，我们会将大部分列表设计为可以垂直滚动或者水平滚动的视图。在 React Native 中 ScrollView 组件是所有列表组件的基础，用来创建一个可以滚动的视图组件，支持垂直或者水平方向的滚动。

6.15.1　ScrollView 的基础用法

ScrollView 组件的设计是为了扩展视图的显示尺寸。因此若要 ScrollView 组件可以正确计算出可以滚动的范围，必须为其指定一个固定的宽高，或者其父视图有固定的尺寸并使用 flex 来指定 ScrollView 的布局尺寸。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ScrollViewDemo.js 的文件。下面的示例代码是最基础的 ScrollView 的用法：

```
import React,{Component} from 'react';
import {ScrollView,Text} from 'react-native';
export default class ScrollViewDemo extends Component{
    render(){
        return(
            <ScrollView style={{flex:1}}>
                <Text style={textStyle}>
                    Hello world!
                </Text>
                <Text style={textStyle}>
                    Hello world!
                </Text>
                <Text style={textStyle}>
                    Hello world!
                </Text>
                <Text style={textStyle}>
                    Hello world!
                </Text>
                <Text style={textStyle}>
                     Hello world!
```

```
                </Text>
            </ScrollView>
        );
    }
}
let textStyle = {
    backgroundColor:'red',
    fontSize:40,
    textAlign:'center',
    marginTop:100
}
```

修改 index.ios.js 与 index.Android.js 文件后运行工程，效果如图 6-29 所示。上下拖拽屏幕可以看到视图的滚动效果。

ScrollView 组件默认支持的是屏幕竖直方向的滚动，其中子视图的布局是竖直排列的，当然也可以是子视图水平排列并且水平滚动的 ScrollView 组件。

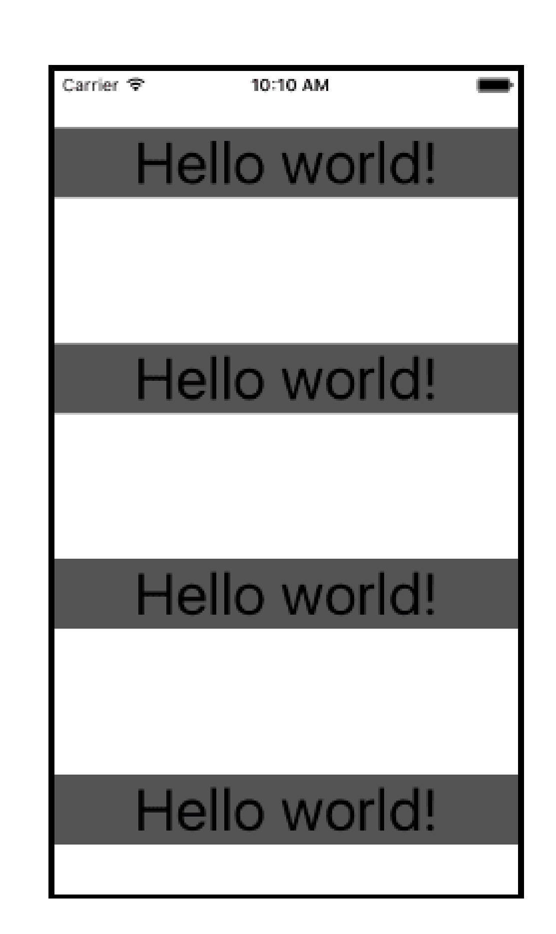


图 6-29 ScrollView 组件

6.15.2 ScrollView 常用属性解析

首先所有 View 组件可用的属性在 ScrollView 组件中都是可以使用的，包括 View 组件所有可用的 Style 风格属性，本节就不再一一列举了。ScrollView 不仅可以进行视图滚动的交互，通过属性的设置也可以实现内容视图缩放的交互。ScrollView 组件常用属性如表 6-29 所示。

表 6-29 ScrollView 组件常用属性

属性名	解释	平台	值类型或可选值
horizontal	设置滚动视图是否水平滚动模式	通用	布尔值
contentContainerStyle	在滚动视图中，所有的子视图都会被添加到一个内容视图容器里，这个属性设置内容视图容器的风格	通用	同 View 组件的 Style 属性可设置的风格对象
keyboardDismissMode	设置当有键盘弹出时，拖拽视图是否隐藏键盘	通用	枚举字符串 • none：拖拽时不隐藏键盘 • on-dray：开始拖拽时隐藏键盘 • interactive：向下拖拽隐藏键盘，向上拖拽会恢复键盘(仅 iOS 平台有效)

（续表）

属性名	解释	平台	值类型或可选值
keyboardShouldPersistTaps	设置当有键盘弹出时，单击是否隐藏键盘	通用	枚举字符串 • never：单击 TextInput 之外的组件会收起键盘 • always：键盘不会收起 • handled：单击事件被子组件捕获时键盘不会收起，单击事件没有被捕获键盘收起
onContentSizeChange	当 ScrollView 组件可滚动尺寸发生变化时调用的回调	通用	函数，会传入如下格式的参数：(contentWidth, contentHeight)
onScroll	组件滚动时调用的回调	通用	函数
refreshControl	设置下拉刷新组件	通用	指定的 RefreshControl 组件
removeClippedSubviews	设置屏幕之外的子视图是否被移除	通用	布尔值
showsHorizontalScrollIndicator	设置是否显示水平方向的滚动条	通用	布尔值
showsVerticalScrollIndicator	设置是否显示竖直方向的滚动条	通用	布尔值
pagingEnabled	是否开启分页效果	通用	布尔值
scrollEnabled	设置是否可以滚动	通用	布尔值
alwaysBounceHorizontal	设置水平方向是否始终显示弹性回拉效果	iOS	布尔值
alwaysBounceVertical	设置竖直方向是否始终显示弹性回拉效果	iOS	布尔值
automaticallyAdjustContentInsets	设置当滚动视图布局在导航栏后面时是否自动调整布局	iOS	布尔值
bounces	是否有弹性回拉效果	iOS	布尔值
bouncesZoom	设置是否有缩放回弹效果	iOS	布尔值
canCancelContentTouches	设置当子控件接收事件时是否取消滚动事件	iOS	布尔值
centerContent	设置内容视图是否居中	iOS	布尔值
contentInset	设置内容视图的边距	iOS	如下格式的对象：{top: number, left: number, bottom: number, right: number}
contentOffset	手动设置滚动视图的显示位置	iOS	如下格式的对象：{x: 0, y: 0}
decelerationRate	设置滚动视图的减速速度	iOS	枚举字符串 • Normal：正常 • Fast：快速
indicatorStyle	设置滚动条的风格	iOS	枚举字符串： • default：默认 • black：黑色 • white：白色

（续表）

属性名	解释	平台	值类型或可选值
maximumZoomScale	设置缩放的最大比例	iOS	数值
minimumZoomScale	设置缩放的最小比例	iOS	数值
onScrollAnimationEnd	滚动动画结束之后调用的回调	iOS	函数
scrollEventThrottle	这个属性设置在 ScrollView 组件滚动过程中滚动事件的调用频率	iOS	数值，表示每秒调用多少次
scrollIndicatorInsets	设置滚动条距离组件的边距	iOS	如下格式对象：{top: number, left: number, bottom: number, right: number}
scrollsToTop	当这个值设置为 true 时，单击状态栏会直接滚动到 ScrollView 组件的顶部	iOS	布尔值
snapToInterval	这个属性设置后，滚动视图会停留在其所设置的数值的整数倍位置上	iOS	数值
zoomScale	设置滚动视图的初始缩放比例	iOS	数值，默认为 1
snapToAlignment	当使用 ScrollView 的方法进行滚动区域的设置时，这个属性设置停留位置的对齐模式	iOS	枚举字符串 • start：停留位置与 ScrollView 组件的起始位置对齐（左或上） • center：停留中间 • end：停留底部

6.15.3 手动设置 ScrollView 组件的滚动位置

ScrollView 中提供了两个方法供开发者手动对 ScrollView 组件滚动位置进行设置，示例代码如下：

```
render(){
    return(
        <ScrollView style={{flex:1}}
         contentContainerStyle={{backgroundColor:"blue"}}
         pagingEnabled={true}
         stickyHeaderIndices={[0]}
         snapToAlignment={'center'}
         ref='scrollView'>
       <Text style={textStyle}
        onPress={()=>{
                    this.refs.scrollView.scrollTo({x:0,y:100,animated:true});
                }}>
            Hello world!
       </Text>
```

```
            <Text style={textStyle}
             onPress={()=>{
                         this.refs.scrollView.scrollToEnd({animated: true});
                     }}>
                Hello world!
            </Text>
            <Text style={textStyle}>
                Hello world!
            </Text>
            <Text style={textStyle}>
                Hello world!
            </Text>
            <Text style={textStyle}>
                Hello world!
            </Text>
        </ScrollView>
    );
```

scrollTo()方法中需要传入要滚动到的位置坐标，animated 参数为布尔值，决定滚动过程是否带动画效果。scrollToEnd()方法用来直接滚动到滚动视图的末尾，其中也可以通过设置 animated 参数来决定是否带动画。

6.16　ListView 列表组件的应用

通过前面的学习，你已经掌握了不少 React Native 中独立组件的应用，本节将要介绍的 ListView 组件要比前面学习的组件更加复杂。在实际开发中，ListView 组件的应用十分广泛。

ListView 组件是一个垂直的滚动列表，你可以自定义列表中每一行数据载体视图的具体样式。和前面所学组件不同的是，ListView 组件需要通过 DataSource 数据源来进行渲染。

6.16.1　使用 DataSource 渲染 ListView 视图

在 React Native 中有定义 ListViewDataSource 这样一个类，专门用来创建用于渲染 ListView 的数据源。首先在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ListViewDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {ListView,Text} from 'react-native';
export default class ListViewDemo extends Component{
    constructor(props){
        super(props);
        this.dataSource = new ListView.DataSource({
```

```
            rowHasChanged:(r1,r2)=>r1!==r2,
            sectionHeaderHasChanged:(s1,s2)=>s1!==s2,
            getRowData:(data,sID,rID)=>{
                console.log(data,sID,rID);
                return data[sID][rID];
            },
            getSectionHeaderData:(data,sID)=>{
                return "分区"+sID;
            }
        });
        this.dataSource=this.dataSource.cloneWithRows([
            "数据","数据",
            "数据","数据",
            "数据","数据"
        ])
    }
    render(){
        return(
            <ListView dataSource={this.dataSource}
             renderRow={(rowData) => {
                 return(<Text style={{lineHeight:30,marginTop:30,
backgroundColor:'red',textAlign:'center'}}>{rowData}</Text>);
             }}
             renderSectionHeader={(headerData,sID)=>{
                 return(<Text style={{lineHeight:30,marginTop:30,
backgroundColor:'green',textAlign:'center'}}
    onPress={()=>{
        let rows = this.dataSource.getRowCount();
        let secs = this.dataSource.getRowAndSectionCount();
        console.log(rows,secs);
    }}>{headerData}</Text>);
            }}/>
        );
    }
}
```

修改 index.ios.js 与 index.Android.js 文件后，运行工程，效果如图 6-30 所示。

在使用 ListView 组件时，有两个属性是必须提供的：dataSource 与 renderRow。dataSource 属性是列表数据的提供者，renderRow 属性用来设置 ListView 中每一行的具体视图。上面示例代码中还设置了 renderSectionHeader 属性，这个属性用来设置每个分区的标题视图。

抽 丝 剥 茧

ListView 可以进行分区，简单理解，分区就是将数据进行分组，每个分组都可以添加一个标题视图。例如，手机上的联系人应用，通常联系人列表会以姓名首字母进行分组排序。

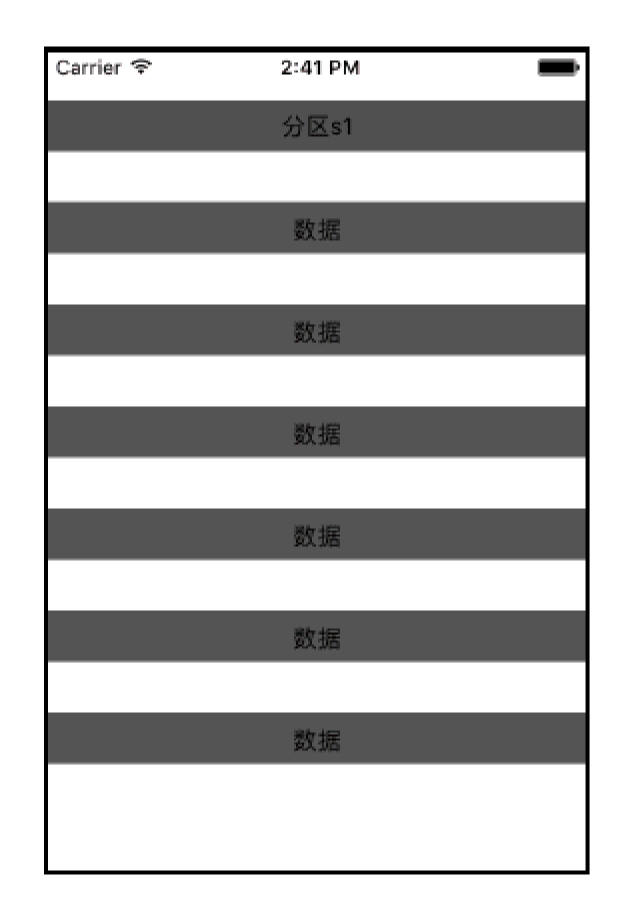


图 6-30　ListView 组件效果

要彻底掌握 ListView 组件的应用，首先需要深入理解 DataSource 的意义与用法。DataSource 的构造函数中需要传入一个对象参数，此对象可以进行设置的属性如表 6-30 所示。

表 6-30　DataSource 构造函数中对象参数的属性

属性名	解释	平台	备注
getRowData	设置获取行数据的函数	通用	此函数会传入 3 个参数，依次为： • data：原始数据 • sectionId：分区 ID • rowId：行 ID 返回数据用于行渲染
getSectionHeaderData	设置获取分区头数据的函数	通用	此函数会传入两个参数，依次为： • data：原始数据 • sectionId：分区 ID 返回数据用于分区标题视图渲染
rowHasChanged	设置判断行数据改变判定的函数	通用	此函数会传入两个参数，依次为： • r1：数据修改前的行数据 • r2：数据修改后的行数据 返回布尔值表示此行数据是否改变了
sectionHeaderHasChanged	设置分区头数据改变判定的函数	通用	此函数会传入两个参数，依次为： • s1：数据修改前的分区头数据 • s2：数据修改后的分区头数据 返回布尔值表示此分区头数据是否改变了

需要注意，上面列出的 4 个属性都是可选的，如果开发者不提供获取数据的方法，则 ListView 会按默认的规则从原始数据中获取用于渲染的数据，这种情况下要求原始数据必须严格遵守如下 3 种格式中的一种：

- {sectionID1:{rowID1:rowData1,rowID2:rowData2,...},...}
- {sectionID1:[rowData1,rowData2,...],...}
- [[rowData1,rowData2],...]

DataSource 中的 cloneWithRows 方法用来指定原始数据。注意，这个方法会返回一个新的 DataSource 对象，并不会修改原 DataSource 对象。cloneWithRows 方法中可以传入两个参数，第 1 个参数为原始数据 data，第 2 个参数需要是一个数组，数组中对应每一行的 rowId。cloneWithRows 方法在只有一个分区时使用，实际上，在 React Native 内部的实现是创建了一个 sectionId 为 s1 的分区，使用这个方法传入的原始数据实际上会被转换成如下格式:{s1:data}。

与 cloneWithRows 方法对应，DataSource 中还有一个 cloneWithRowsAndSections 方法。这个方法在 ListView 多分区时用来设置原始数据，其中可以传入 3 个参数，依次为原始数据 data、分区 sectionId 数组和每个分区的 rowId（二维数组）。

DataSource 中其他常用方法如表 6-31 所示。

表 6-31　DataSource 的其他方法

函数名	解释	平台	参数
getRowCount	获取总行数	通用	无
getRowAndSectionCount	获取分区数加上总行数	通用	无
rowShouldUpdate(sectionIndex, rowIndex)	获取某行是否需要刷新	通用	sectionIndex：分区下标，从 0 开始 rowIndex：行下标，从 0 开始
getRowIDForFlatIndex(index)	通过给定索引值获取 rowId	通用	index：索引值
getSectionIDForFlatIndex(index)	通过给定索引值获取 sectionId	通用	index：索引值
getSectionLengths	获取每个分区的行数	通用	返回数组，里面存放每个分区的行数
sectionHeaderShouldUpdate(sectionIndex)	获取某个分区头是否需要刷新	通用	sectionIndex：分区下标

上面 ListView 的示例代码只是最简单的列表创建，通常情况下，分区可能有多个，每一行的数据也应该各有不同，将代码修改如下：

```
constructor(props){
    super(props);
    this.dataSource = new ListView.DataSource({
        rowHasChanged:(r1,r2)=>r1!==r2,
        sectionHeaderHasChanged:(s1,s2)=>s1!==s2,
        getRowData:(data,sID,rID)=>{
            console.log(data,sID,rID);
            return "第"+rID+"行"+data[sID][rID];
        },
        getSectionHeaderData:(data,sID)=>{
            return "分区"+sID;
        }
    });
    this.dataSource=this.dataSource.cloneWithRowsAndSections([[
```

```
            "数据","数据",
            "数据","数据",
            "数据","数据"
        ],[
            "数据","数据",
            "数据","数据",
        ]])
    }
```

运行工程，效果如图 6-31 所示。



图 6-31　多分区的 ListView

6.16.2　ListView 属性方法解析

理解了 DataSource 之后，学习 ListView 将如鱼得水。ListView 是扩展于 ScrollView 的组件，因此所有 ScrollView 可用的属性，ListView 组件都可以使用。ListView 常用属性如表 6-32 所示。

表 6-32　ListView 常用属性

属性名	解释	平台	值类型或可选值
dataSource	设置数据源	通用	ListView.DataSource 实例
initialListSize	设置挂载完成时加载的行数，这个属性设置得当可以消除下载 ListView 时的闪屏	通用	数值
onChangeVisibleRows	当可见行发生改变时调用的回调	通用	函数，会传入如下格式的参数 (visibleRows, changedRows) 其中 visibleRows 和 changedRows 都是如下对象：{ sectionID: { rowID: true|false}} 布尔值代表此行是否可见

（续表）

属性名	解释	平台	值类型或可选值
onEndReachedThreshold	设置调用 onEndReached 回调的临界值	通用	数值
onEndReached	当所有的数据渲染过并且列表距离底部不足 onEndReached-Threshold 设置的临界值时调用的回调	通用	函数
pageSize	每帧渲染的行数	通用	数值
removeClippedSubviews	非可见的行是否被移除	通用	布尔值
renderFooter	设置列表的尾视图组件	通用	函数，需要返回组件实例
renderHeader	设置列表的头视图组件	通用	函数，需要返回组件实例
renderRow	设置每行视图组件	通用	函数，传参格式如下： (rowData, sectionID, rowID, highlightRow) 需要返回组件实例
renderScrollComponent	设置 ScrollView 容器	通用	函数，需要返回 ScrollView 组件作为容器，如果不设置这个属性，将默认创建一个 ScrollView 作为容器
renderSectionHeader	设置每个分区的标题视图	通用	函数，传参格式如下： (sectionData, sectionID) 需要返回一个组件实例
renderSeparator	设置分割线	通用	函数，传参格式如下： (sectionID, rowID, adjacentRowHighlighted) adjacentRowHighlighted 参数表示临近行是否高亮，需要返回组件实例
scrollRenderAheadDistance	设置当某一行距离屏幕多少距离时就开始提前渲染	通用	数值

ScrollView 组件中所有可用的方法在 ListView 中也是同样适用的，例如滚动到某一位置，滚动到末尾等。

博 闻 强 识

ListView 虽然十分强大，但在数据量很大、列表行数很多的情况下仍不能保证良好的内存回收，这往往会造成不可控的内存问题。后面我们会学习 FlatView 组件，它和 ListView 的最大区别就在于内存的管理优化。

6.17　高性能列表组件 FlatList

FlatList 组件是在 React Native 0.43 版本之后引入的新列表组件，优化了 ListView 的加载性能并且将列表的渲染简洁化，去掉了通过数据源 DataSource 的渲染方式，直接使用简单数组来进行数据的获取。对于功能上来说，FlatList 和 ListView 基本相似，FlatList 组件不能进行分区但是可以进行多列布局，ListView 不能进行多列布局但是可以分区。

6.17.1　创建一个简单的 FlatList 列表视图

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 FlatListDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {FlatList,Text,View} from 'react-native';
export default class FlatListDemo extends Component{
    constructor(props){
    super(props);
     this.dataSource = [
     {
         key:1,
        value:"数据"
     },
     {
        key:2,
        value:"数据"
    },
     {
         key:3,
         value:"数据"
     },
     {
         key:4,
         value:"数据"
     },
     {
         key:5,
         value:"数据"
     }
     ];
     }
     render(){
```

```
            return(
                <FlatList data={this.dataSource}
                renderItem={(item)=>{
                    return(<Text style={{marginLeft:20,backgroundColor:'red',
marginTop:10,textAlign:'center',lineHeight:30,fontSize:22}}>item</Text>);
                }}
                ItemSeparatorComponent={()=>{
                    return(
                    <View style={{height:2,backgroundColor:'black'}}></View>
                    );
                }}
                ListFooterComponent={()=>{
                    return(
                    <Text style={{backgroundColor:'blue',textAlign:'center',
lineHeight:20,fontSize:18}}>FOOTER</Text>
                );
                }}
                ListHeaderComponent={()=>{
                    return(
                    <Text style={{backgroundColor:'blue',textAlign:'center',
lineHeight:20,fontSize:18}}>HEADER</Text>
                    );
                }}
                columnWrapperStyle={{backgroundColor:'green'}}
                numColumns={3}
                getItemLayout={(dataArray,index)=>{
                    return {length:50,offset:52*index,index:index};
                }}
            />
        );
    }
    }
```

上面代码创建了 3 列的表格视图，修改 index.ios.js 与 index.Android.js 文件后运行工程，效果如图 6-32 所示。

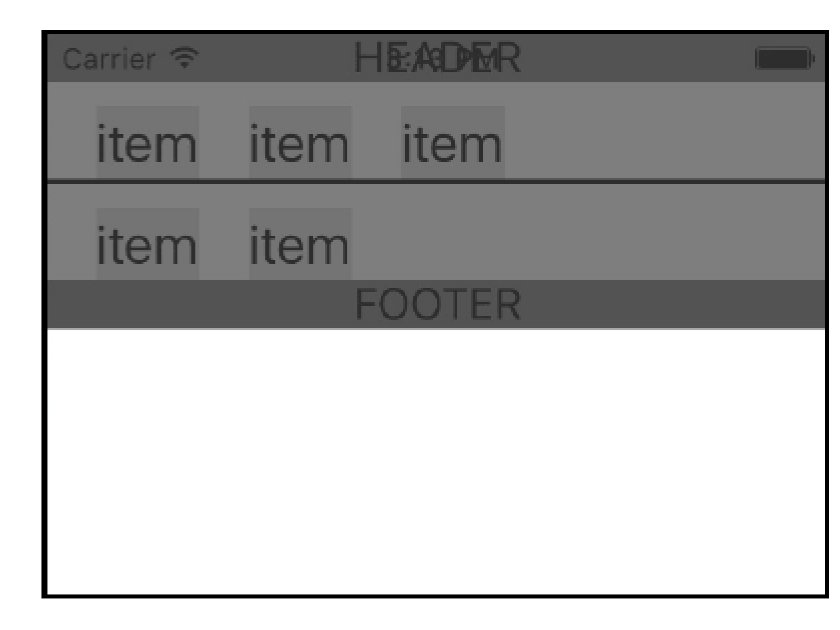


图 6-32　FlatList 组件效果

FlatList 组件常用属性列表如表 6-33 所示。

表 6-33　FlatList 组件常用属性

属性名	解释	平台	值类型或可选值
ItemSeparatorComponent	设置行间分割线	通用	函数，需要返回一个组件
ListFooterComponent	设置列表的尾视图	通用	函数，需要返回一个组件
ListHeaderComponent	设置列表的头视图	通用	函数，需要返回一个组件
columnWrapperStyle	设置行容器样式，当布局模式为多列时可用	通用	style 对象
data	设置数据源，需要为数组对象，数组存放的对象必须包含一个名为 key 的属性	通用	数组对象
getItemLayout	手动设置每一个元素的布局，这样做可以优化性能，减少自适应布局造成的计算成本	通用	函数，会传入如下参数： (data, index) 其中 data 为数据源数组，index 为元素下标 需要返回如下格式的对象： {length: number, offset: number, index: number} 其中 length 表示元素的高，offset 表示元素的位置，index 表示元素的下标
horizontal	设置布局模式是否为水平布局	通用	布尔值，需要注意，如果设置为 true，则不能支持多列模式
keyExtractor	这个属性设置的函数需要返回一个唯一的字符串作为元素的 key 值，如果不实现这个函数，则在数据源中每个数据必须提供 key 属性	通用	函数，会传入如下格式的参数： (item, index) item 为此元素对应的数据，index 为元素下标，需要返回字符串作为此元素的 key
legacyImplementation	是否以 ListView 的实现方式来渲染列表，即不进行性能优化	通用	布尔值
numColumns	设置列数	通用	数值
onEndReached	当列表滚动到距离底部一定距离时调用的函数	通用	函数，参数格式如下： (info: {distanceFromEnd: number}) distanceFromEd 为距底部的距离
onEndReachedThreshold	设置距离底部多少距离触发 onEndReached	通用	数值
onRefresh	设置刷新时回调的方法，当这个属性设置后，FlatList 会自动创建一个刷新组件	通用	函数

（续表）

属性名	解释	平台	值类型或可选值
refreshing	设置刷新状态	通用	布尔值 true：开始刷新 false：结束刷新
onViewableItemsChanged	当元素的可见性发生改变时调动的函数	通用	函数，传参格式如下： ({viewableItems, changed}) viewableItem 为所有可见的元素信息数组，changed 为所有可见状态改变的元素信息数组

6.17.2　FlatList 中常用方法解析

FlatList 组件中也定义了几个常用的实例方法（见表 6-34），直接使用 FlatList 对象来调用即可。

表 6-34　FlatList 组件常用的实例方法

方法名	解释	平台	参数解析
scrollToEnd	滚动到列表底部	通用	可以传入如下格式的对象： {animated:true}表示是否有动画效果
scrollToIndex	滚动到某个元素索引	通用	可以传入如下格式的对象： {animated: true, index: 1, viewPosition: 0} 其中，animated 表示是否有动画效果，index 为元素索引，viewPosition 为以元素的哪个位置为参照点，取值在 0～1 之间，如果设置为 0.5 就表示滚动到指定元素的中间
scrollToOffset	滚动到指定位置	通用	{animated: true, offset: 100}
scrollToItem	滚动到指定的元素	通用	{animated: true, item: object, viewPosition: 1} 其中 item 为元素对象

6.18　分区列表组件 SectionList 的应用

SectionList 组件与 FlatList 组件用法基本一致，SectionList 组件不支持多列布局但是支持对数据进行分区。和 FlatList 一样，SectionList 和 ListView 相比性能更加优化，需要注意，SectionList 组件必须在 React Native 0.43 版本之后使用。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 SectionListDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {SectionList,Text,View} from 'react-native';
export default class SectionDemo extends Component{
```

```
    constructor(props){
        super(props);
        this.sectionSource = [
            {
                data:[
                    {
                        title:"分区 1 数据 1",
                        key:"r1"
                    },
                    {
                        title:"分区 1 数据 2",
                        key:"r2"
                    }
                ],
                key:"s1"
            },
            {
                data:[
                    {
                        title:"分区 2 数据 1",
                        key:"r3"
                    },
                    {
                        title:"分区 2 数据 2",
                        key:"r4"
                    }
                ],
                key:"s2"
            }
        ];
    }
    render(){
        return(
            <SectionList
             renderItem={(data)=>{
                return(<Text style={itemStyle}>{data.item.title}</Text>);
             }}
             sections={this.sectionSource}
             ItemSeparatorComponent={()=>{
                return(<View style={separatorStyle}></View>);
             }}
             ListFooterComponent={()=>{
                return(<Text style={footOrHeadStyle}>FOOTER</Text>);
             }}
             ListHeaderComponent={()=>{
```

```
                    return(<Text style={footOrHeadStyle}>HEADER</Text>);
                 }}
                 SectionSeparatorComponent={()=>{
                    return(<View style={{height:3, backgroundColor:
'green'}}></View>);
                 }}
                 renderSectionHeader={(data)=>{
                    return(<Text style={itemStyle}>{data.section.key}</Text>);
                 }}
                />
            );
        }
    }
    let itemStyle = {
        lineHeight:30,
        backgroundColor:'red',
        textAlign:'center',
        fontSize:22
    }
    let separatorStyle = {
        left:30,
        backgroundColor:'gray',
        height:1
    }
    let footOrHeadStyle = {
        lineHeight:50,
        backgroundColor:'blue',
        textAlign:'center',
        fontSize:24
    }
```

SectionList 采用的数据源也是数组对象，其中第一层为分区数据对象，每个分区对象中必须包含一个 data 属性作为分区中的行数据，key 属性作为分区的唯一标识。行数据对象中也必须包含一个 key 属性作为行唯一标识。运行工程，效果如图 6-33 所示。

SectionList 中的常用属性如表 6-35 所示。



图 6-33　SectionList 组件效果

表 6-35　SectionList 中的常用属性

属性名	解释	平台	值类型或可选值
ItemSeparatorComponent	设置行间分割线，每个分区首行前和末行后不会添加	通用	函数，需要返回组件对象
ListFooterComponent	设置列表的尾视图	通用	函数，需要返回组件对象
ListHeaderComponent	设置列表的头视图	通用	函数，需要返回组件对象
SectionSeparatorComponent	设置分区间的分割线	通用	函数，需要返回组件对象
keyExtractor	这个属性设置的函数需要返回一个唯一的字符串作为行的 key 值	通用	函数，会传入如下格式参数： (item: Item, index: number) 需要返回字符串作为键值
onRefresh	设置刷新调用的回调	通用	函数
refreshing	是否开启刷新	通用	布尔值
renderItem	渲染每行数据	通用	函数，会将行数据以如下格式传入： {item: Item, index: number} 需要返回组件对象
renderSectionHeader	渲染每个分区头视图	通用	会将分区数据以如下格式传入： {section: SectionT} 需要返回组件对象
sections	设置数据源数组	通用	数组，格式如前面的示例代码所示

博 闻 强 识

至此，你已经将 React Native 中所有可能使用到的列表组件学习了一遍，列表组件是开发中非常常用的一种组件，后面的实战过程中，你也会体会到列表组件的精髓所在。

6.19　RefreshControl 刷新组件的应用

FlatList 组件与 SectionList 组件是 React Native 0.43 版本中引入的新组件，内置下拉刷新组件。对于 ScrollView 与 ListView，你可以通过手动设置下拉组件来使其支持下拉刷新功能。

打开 HelloWorld 工程中的 ScrollViewDemo.js 文件，修改如下：

```
import React,{Component} from 'react';
import {ScrollView,Text,RefreshControl} from 'react-native';
export default class ScrollViewDemo extends Component{
    constructor(props){
        super(props);
        this.state = {
            refresh:false
        }
```

```
        }
        render(){
            return(
                <ScrollView style={{flex:1}}
                contentContainerStyle={{backgroundColor:"blue"}}
                pagingEnabled={false}
                stickyHeaderIndices={[0]}
                snapToAlignment={'center'}
                ref='scrollView'
                refreshControl={this.getRefreshControl()}>
                    <Text style={textStyle}
                    onPress={()=>{
                        this.refs.scrollView.scrollTo({x:0,y:100,
animated:true});
                    }}>
                        Hello world!
                    </Text>
                    <Text style={textStyle}
                    onPress={()=>{
                        this.refs.scrollView.scrollToEnd({animated: true});
                    }}>
                        Hello world!
                    </Text>
                    <Text style={textStyle}
                    onPress={()=>{
                        this.setState({
                            refresh:false
                            });
                    }}>
                        Hello world!
                    </Text>
                    <Text style={textStyle}>
                        Hello world!
                    </Text>
                    <Text style={textStyle}>
                        Hello world!
                    </Text>
                </ScrollView>
            );
        }
        getRefreshControl(){
            return(
                <RefreshControl onRefresh={()=>{
                this.setState({
                    refresh:true
```

```
                    });
                    console.log("开始刷新");
                }}
                refreshing={this.state.refresh}
                colors={['red','green']}
                enabled={true}
                progressBackgroundColor="blue"
                size={RefreshControl.SIZE.LARGE}
                progressViewOffset={100}
                tintColor='red'
                title="refreshing"/>
            );
        }
    }
    let textStyle = {
        backgroundColor:'red',
        fontSize:40,
        textAlign:'center',
        marginTop:100,
    }
```

运行工程，在 Android 与 iOS 模拟器上运行工程，通过下拉可以看到下拉刷新组件的效果。RefreshControl 中很多属性都是分平台的，常用属性如表 6-36 所示。

表 6-36　RefreshControl 常用属性

属性名	解释	平台	值类型或可选值
onRefresh	刷新组件激活时的回调函数	通用	函数
refreshing	设置是否开启刷新状态	通用	布尔值
colors	设置绘制刷新组件的颜色数组	Android	数组
enabled	设置刷新组件是否启用	Android	布尔值
progressBackgroundColor	设置刷新组件的背景颜色	Android	特定的颜色字符串
size	设置刷新组件的尺寸	Android	在如下变量中选择： • RefreshControl.SIZE.DEFAULT: 默认大小 • RefreshControl.SIZE.LARGE：大尺寸
progressViewOffset	设置刷新组件竖直方向上的位置	Android	数值
tintColor	设置刷新组件颜色	iOS	特定的颜色字符串
title	设置刷新组件显示的文字	iOS	字符串

ListView 组件是扩展于 ScrollView 组件的，因此在 ListView 中，RefreshControl 组件的用法和在 ScrollView 中的用法一致。你可以尝试修改一下 ListViewDemo.js 文件，使我们自定义的 ListView 视图支持下拉刷新。

第 7 章

React Native 独立组件高级篇

通过第 6 章的学习，你已经掌握了许多 React Native 中跨平台的独立组件。然而 iOS 平台与 Android 平台毕竟差异很大，在 React Native 中还定义了许多专用于某种平台的组件。本章我们将主要介绍这类组件的应用。

一款完整的应用程序几乎不可能只有一个界面，本章你还将学习如何在 React Native 中管理界面路由，实现界面的跳转切换。通过本章的学习，相信你的 React Native 开发能力会再上一个台阶。

7.1 时间选择器 DatePickerIOS 组件的应用

在前面章节中，已经学习了选择器 Picker 组件。其实在 iOS 平台上，React Native 还提供了一个专门用来选择日期时间的组件 DatePickerIOS。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 DatePickerIOSDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {DatePickerIOS} from 'react-native';
export default class DatePickerIOSDemo extends Component{
    constructor(props){
        super(props);
        this.state = {
            date:new Date()
        }
    }
    render(){
        return(
            <DatePickerIOS style={{width:300,height:200,left:10,top:30}}
            date={this.state.date}
```

```
                mode={'time'}
                onDateChange={(newDate)=>{
                    this.setState({
                        date:newDate
                    });
                }}/>
            );
        }
    }
```

DatePickerIOS 组件有 3 种模式，分别为 date、time 和 datetime。修改 index.ios.js 文件后运行工程，效果如图 7-1 所示。

需要注意，DatePickerIOS 组件是一个受控组件，用户的操作无法改变 DatePickerIOS 组件的选择值，开发者在接收到用户选择回调后，需要手动修改 DatePickerIOS 组件的 date 属性。DatePickerIOS 组件扩展于 View 组件，因此 View 组件的所有属性都支持，其他常用属性如表 7-1 所示。

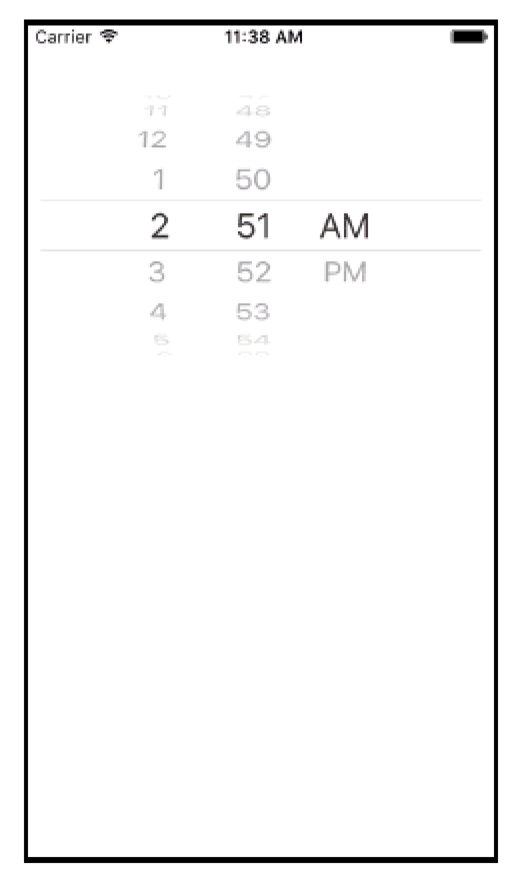


图 7-1　DatePickerIOS 组件样式

表 7-1　DatePickerIOS 组件的其他常用属性

属性名	解释	平台	值类型或可选值
date	当前选中的日期	iOS	Date 对象
maximumDate	可选的最大日期	iOS	Date 对象
minimumDate	可选的最小日期	iOS	Date 对象
mimuteInterval	可选的时间分钟间隔	iOS	可选数值为 1、2、3、4、5、6、10、12、15、20、30
mode	选择器组件模式	iOS	枚举字符串 • date：日期模式 • time：时间模式 • datetime：日期时间模式
onDateChange	当用户修改日期时回调的函数	iOS	函数，会传入用户选择的 Date 日期对象
timeZoneOffsetInMinutes	设置时区差，单位是分钟	iOS	数值，用来指定时区的分钟差，例如东八区可以设置 8*60

7.2　DrawerLayoutAndroid 抽屉组件的应用

在安卓设备上，很容易见到各式各样的抽屉视图。抽屉视图常常可以通过在设备屏幕边缘滑

动手势来打开，在 React Native 中，提供了 DrawerLayoutAndroid 组件来创建抽屉视图。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 DrawerLayoutAndroidDemo.js 的文件，在其中编写如下代码：

```
    import React,{Component} from 'react';
    import {DrawerLayoutAndroid,View,Text} from 'react-native';
    export default class DrawerLayoutAndroidDemo extends Component{
        render(){
            return(
            <DrawerLayoutAndroid
            drawerWidth={150}
            drawerPosition={DrawerLayoutAndroid.positions.Left}
            renderNavigationView={()=>{
                return(
                    <View style={{flex: 1, backgroundColor: '#fff'}}>
                        <Text style={{margin: 10, fontSize: 15, textAlign:
'left'}}>抽屉视图</Text>
                </View>
                );
            }}>
                <View style={{flex: 1, alignItems: 'center'}}>
                <Text style={{margin: 10, fontSize: 15, textAlign: 'right'}}>
Hello</Text>
                <Text style={{margin: 10, fontSize: 15, textAlign: 'right'}}>
World!</Text>
                </View>
            </DrawerLayoutAndroid>
            );
        }
    }
```

DrawerLayoutAndroid 组件是一个容器组件，其中的子组件会渲染在显示内容中，renderNavigationView 属性用来设置抽屉视图，修改 index.android.js 文件后，运行工程，效果如图 7-2 所示。

通过在屏幕上使用左划和右划手势，可以对抽屉进行打开与收起操作。DrawerLayoutAndroid 组件扩展于 View 组件，因此可以使用所有 View 组件可用的属性。DrawerLayoutAndroid 组件中的常用属性如表 7-2 所示。

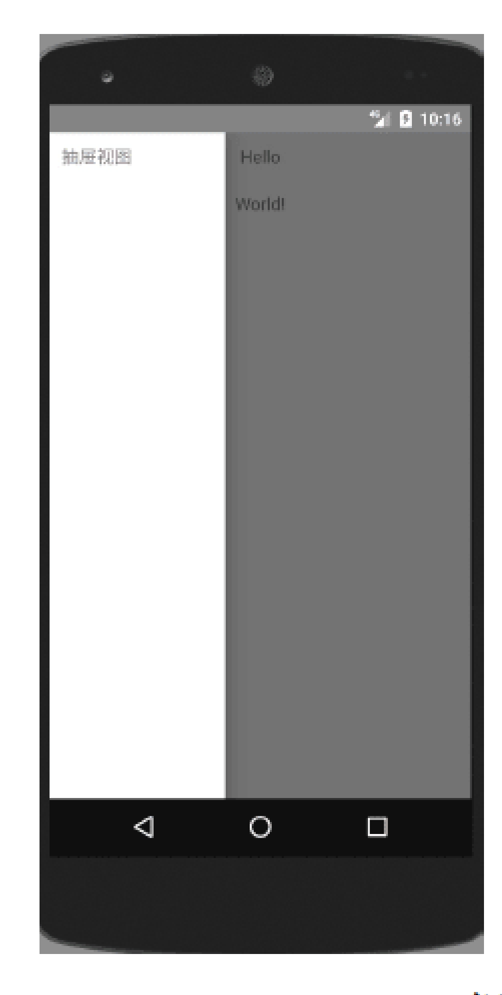


图 7-2 DrawerLayoutAndroid 组件效果

表 7-2　DrawerLayoutAndroid 组件常用属性

属性名	解释	平台	值类型或可选值
keyboardDismissMode	设置在拖拽过程中是否隐藏	Android	枚举字符串 none：不隐藏键盘 on-drag：拖拽时隐藏
drawerPosition	设置抽屉的弹出方向	Android	有如下两种可选值： • DrawerConsts.DrawerPosition.Left：从左侧滑出 • DrawerConsts.DrawerPosition.Right：从右侧滑出
drawerWidth	设置抽屉宽度	Android	数值
drawerLockMode	设置抽屉锁定状态，设置后，只能够通过函数方法来开关，不能通过手势开关	Android	枚举字符串： • unlocked：不锁定 • locked-closed：关闭 • locked-open：开启
onDrawerSlide	在用户滑动抽屉时会不断调用此回调	Android	函数
onDrawerStateChanged	当抽屉状态发生改变时调用的回调	Android	函数，会传入如下 3 种状态之一： • idle：空闲状态，无任何交互 • dragging：拖拽中 • settling：停靠中
onDrawerClose	当抽屉关闭时调用的回调	Android	函数
renderNavigationView	用来渲染抽屉视图	Android	函数，需要返回组件
statusBarBackgroundColor	设置状态栏背景色	Android	特定颜色字符串

可以使用如表 7-3 所示的方法通过代码来实现抽屉的开关。

表 7-3　实现抽屉开关的方法

方法名	解释	平台	参数
openDrawer	打开抽屉	Android	无
closeDrawer	关闭抽屉	Android	无

7.3　进度条组件的应用

无论在 iOS 平台还是 Android 平台都有进度条这一原生控件，在进行数据加载、网络请求、音视频播放等任务时，进度条都起到了不可或缺的提示作用。然而在 React Native 中，进度条组件却不是跨平台的，针对 iOS 与 Android 平台，React Native 分别提供了用于创建进度条的组件。我们可以通过一些兼容平台技巧来同时在双平台上进行进度条组件的应用。

7.3.1 通过文件名分平台加载组件

React Native 支持通过规范的命名文件来使其自适应平台加载代码。前面我们编写的文件都是通用格式的文件，如果需要只在 iOS 平台加载，需要添加.ios 作为后缀名，同样如果需要只在 Android 平台加载，则需要添加.android 作为后缀名。在 React Native 中，iOS 平台的进度条组件定义为 ProgressViewIOS，Android 平台定义的进度条组件为 ProgressBarAndroid。

在 HelloWorld 工程的 Demo 文件夹下面新建两个文件，分别命名为 ProgressBar.ios.js 与 ProgressBar.android.js。当你在其他文件中引用 ProgressBar 文件时，系统会自动判断平台取合适的 JavaScript 文件。将 ProgressBar.ios.js 文件实现如下：

```
import React,{Component} from 'react';
import {ProgressViewIOS} from 'react-native';
export default class ProgressBar extends Component{
    render(){
        return(
            <ProgressViewIOS style={{top:100}}
            progressTintColor='red'
            progressViewStyle={'default'}
            progress={0.5}
            />
        );
    }
}
```

将 ProgressBar.android.js 文件实现如下：

```
import React,{Component} from 'react';
import {ProgressBarAndroid,View} from 'react-native';
export default class ProgressBar extends Component{
    render(){
        return(
            <ProgressBarAndroid style={{top:100}}
            color='red'
            styleAttr={"Horizontal"}
            progress={0.5}
            indeterminate={false}/>
        );
    }
}
```

将 index.ios.js 与 index.android.js 文件修改如下：

```
import React, { Component } from 'react';
import {
  AppRegistry
} from 'react-native';
```

```
import ProgressBar from './Demo/ProgressBar';
export default class HelloWorld extends Component {
  render() {
    return (<ProgressBar />);
  }
}
AppRegistry.registerComponent('HelloWorld', () => HelloWorld)
```

在两个平台上运行工程，效果分别如图 7-3 与图 7-4 所示。

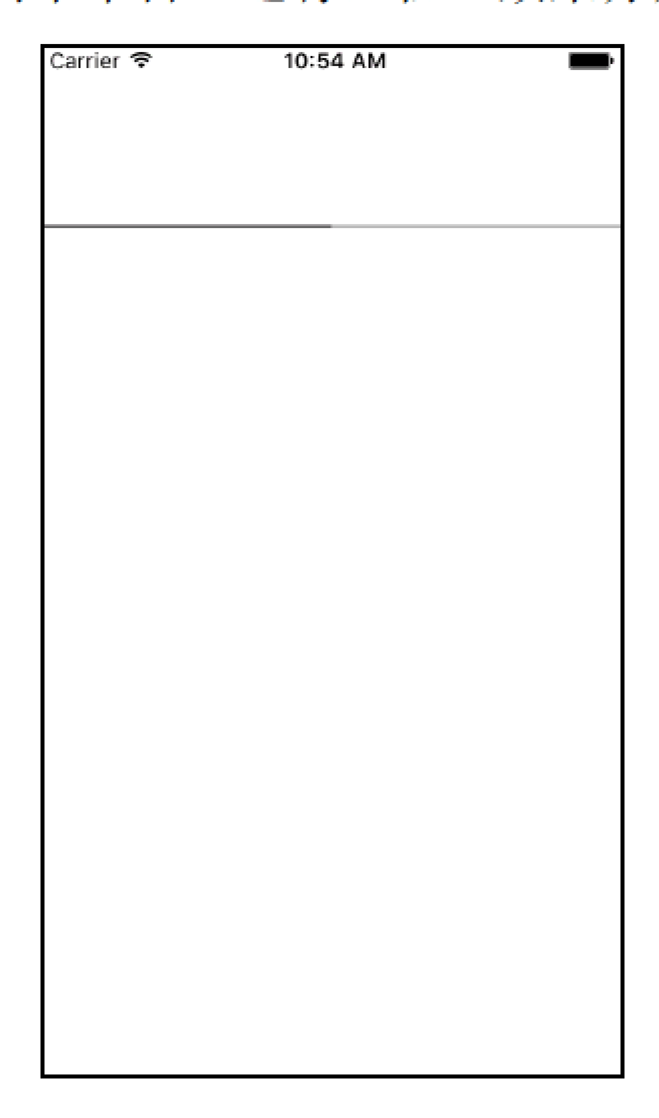


图 7-3　iOS 平台下的进度条组件

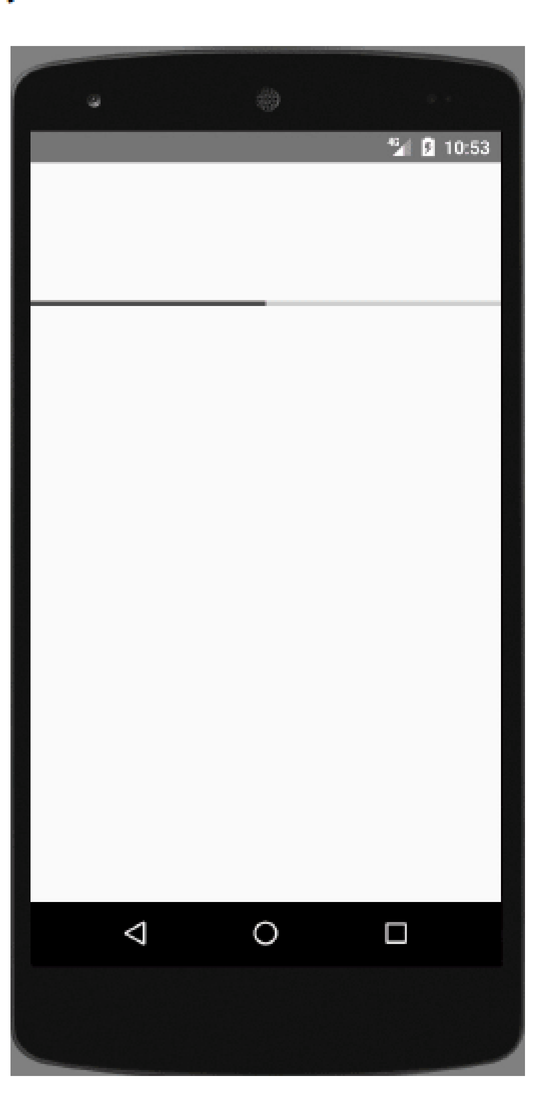


图 7-4　Android 平台下的进度条组件

7.3.2　ProgressBarAndroid 组件常用属性

ProgressBarAndroid 组件是 Android 平台上的进度条组件，扩展于 View 组件并且支持多种风格样式，除了可以使用所有 View 组件可用的属性外，其他常用属性如表 7-4 所示。

表 7-4　ProgressBarAndroid 组件其他常用属性

属性名	解释	平台	值类型或可选值
color	进度条颜色	Android	特定的颜色字符串
progress	当前进度，需要当 indeterminate 设置为 false 并且 styleAttr 为 Horizontal 风格时使用	Android	数值（0～1 之间）
indeterminate	是否显示不确定的进度	Android	布尔值
styleAttr	设置进度条风格	Android	枚举字符串 • Horizontal：水平 • Small：小圆圈 • Large：大圆圈 • Inverse：逆序圆圈 • SmallInverse：小逆序圆圈 • LargeInverse：大逆序圆圈

7.3.3 ProgressViewIOS 组件常用属性

ProgressViewIOS 组件用于在 iOS 平台上创建进度条，其同样也是扩展于 View 组件，因此可以使用所有 View 组件的属性，除此之外，常用属性如表 7-5 所示。

表 7-5 ProgressViewIOS 组件常用属性

属性名	解释	平台	值类型或可选值
progress	当前进度值	iOS	数值（0～1）
progressImage	设置进度条图片	iOS	图片素材
progressTintColor	设置进度条颜色	iOS	特定的颜色字符串
progressViewStyle	设置进度条风格	iOS	枚举字符串： • default • bar
trackImage	设置进度条未走过进度的图片	iOS	图片素材
trackTintColor	设置进度条未走过进度的颜色	iOS	特定的颜色字符串

7.4 SegmentedControlIOS 组件的应用

SegmentedControl 是 iOS 平台上的一个原生控件，又称为分段控制器。SegmentedControlIOS 组件是 React Native 对 iOS 平台中分段控制器的包装。它常用在进行模块选择的功能中。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 SegmentedControlIOSDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {SegmentedControlIOS} from 'react-native';
export default class SegmentedControlIOSDemo extends Component{
    render(){
        return(
            <SegmentedControlIOS
            style={{top:100}}
            onChange={()=>{
                console.log("onChange");
            }}
            onValueChange={(value)=>{
                console.log(value);
            }}
            selectedIndex={0}
            tintColor='red'
```

```
                values={['one','two','three']}/>
            );
        }
    }
```

修改 index.ios.js 文件后，运行工程，效果如图 7-5 所示。

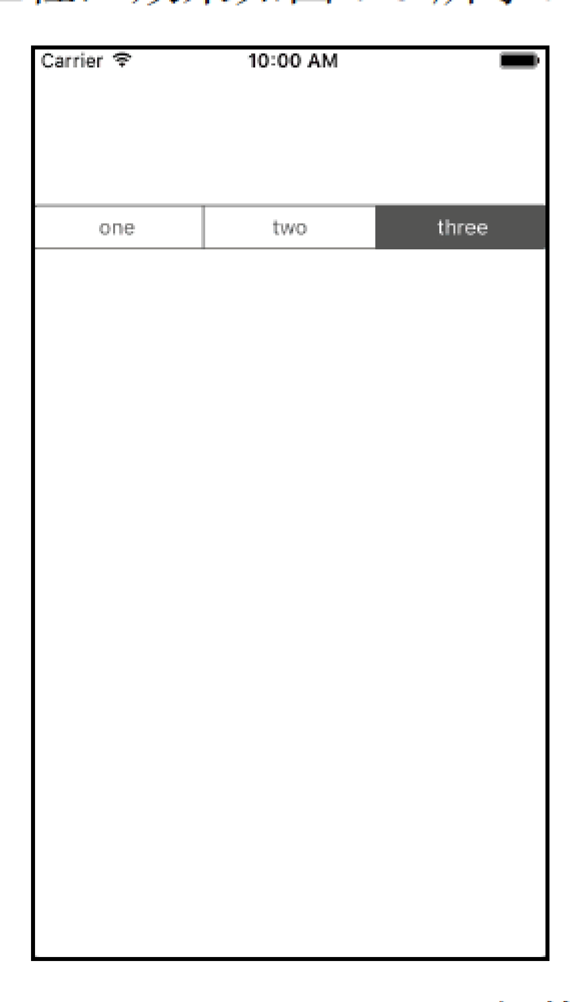


图 7-5　SegmentedControlIOS 组件效果

当用户单击分段控制器中的某一段时可以实现选中切换。SegmentedControlIOS 组件扩展于 View 组件，因此可以使用所有 View 组件可用的属性，除此之外的常用属性如表 7-6 所示。

表 7-6　SegmentedControlIOS 组件常用属性

属性名	解释	平台	值类型或可选值
enabled	设置组件是否可以交互	iOS	布尔值
momentary	设置是否不保持选中状态，如果设置为 true，则分段控制器不会保持选中状态，但相应回调依然会调用	iOS	布尔值
onChange	当用户单击了某段会调用的回调	iOS	函数
onValueChange	当用户单击某段时调用，会将选中的值作为参数传入	iOS	函数
selectedIndex	设置初始选中的段下标	iOS	数值
tintColor	设置分段控制器颜色	iOS	特定的颜色字符串
values	设置每段的值，会显示为标题	iOS	数组，其中的元素需要为字符串类型

7.5　Android 平台上的工具条组件

React Native 中提供了 ToolbarAndroid 组件，用来在 Android 平台上创建工具条。ToolbarAndroid 组件十分灵活，不仅提供了丰富的属性供我们对工具条进行定制，还可以完全定义自己的工具条组件。

在 HelloWorld 工程的 Demo 文件夹下面新建一个命名为 ToolbarAndroidDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {ToolbarAndroid,View} from 'react-native';
export default class ToolbarAndroidDemo extends Component{
    render(){
        return(
            <ToolbarAndroid
            style={{height:56,backgroundColor:'green'}}
            logo={require('./../source/logo.png')}
            navIcon={require('./../source/setting.png')}
            overflowIcon={require('./../source/setting.png')}
            subtitle='React Native'
            subtitleColor='blue'
            title='ToolbarAndroid'
            titleColor='red'
            actions={[
                {
                    title:'setting1',
                    icon:require('./../source/timg.jpeg'),
                    show:'always',
                    showWithText:true
                },
                {
                    title:'setting2',
                    icon:require('./../source/timg.jpeg'),
                    show:'ifRoom',
                    showWithText:true
                },
                {
                    title:'setting3',
                    icon:require('./../source/timg.jpeg'),
                    show:'never',
                    showWithText:true
                },
            ]}
            onActionSelected={(position)=>{
                console.log(position);
            }}
            onIconClicked={()=>{
                console.log('icon');
            }}
            />
        );
    }
}
```

ToolbarAndroid 组件大致分为 3 个部分：导航图标部分，标题部分和功能列表部分。其中，导航图标部分在左，标题部分在中间，功能列表部分在右侧。功能列表也支持进行折叠显示。修改 index.android.js 文件后，运行工程，效果如图 7-6 所示。

上面的示例代码使用的是默认的工具条样式，是使用单标签创建的。同样，也可以使用双标签来创建完全自定的工具条，示例如下：

```
                <ToolbarAndroid style={{backgroundColor:'red',height:50,
marginTop:50}}
                logo={require('./../source/logo.png')}>
                    <Switch />
                    <Text>Switch</Text>
                </ToolbarAndroid>
```

运行工程效果如图 7-7 所示。

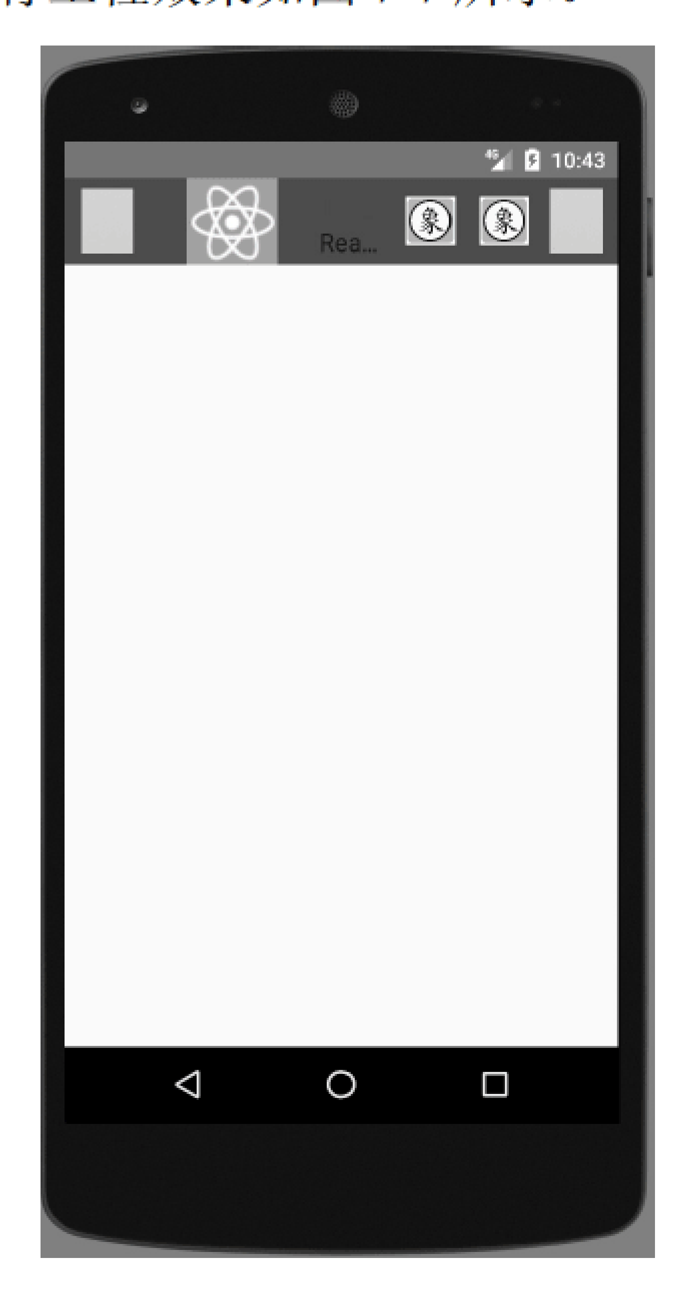



图 7-6　ToolbarAndroid 组件效果

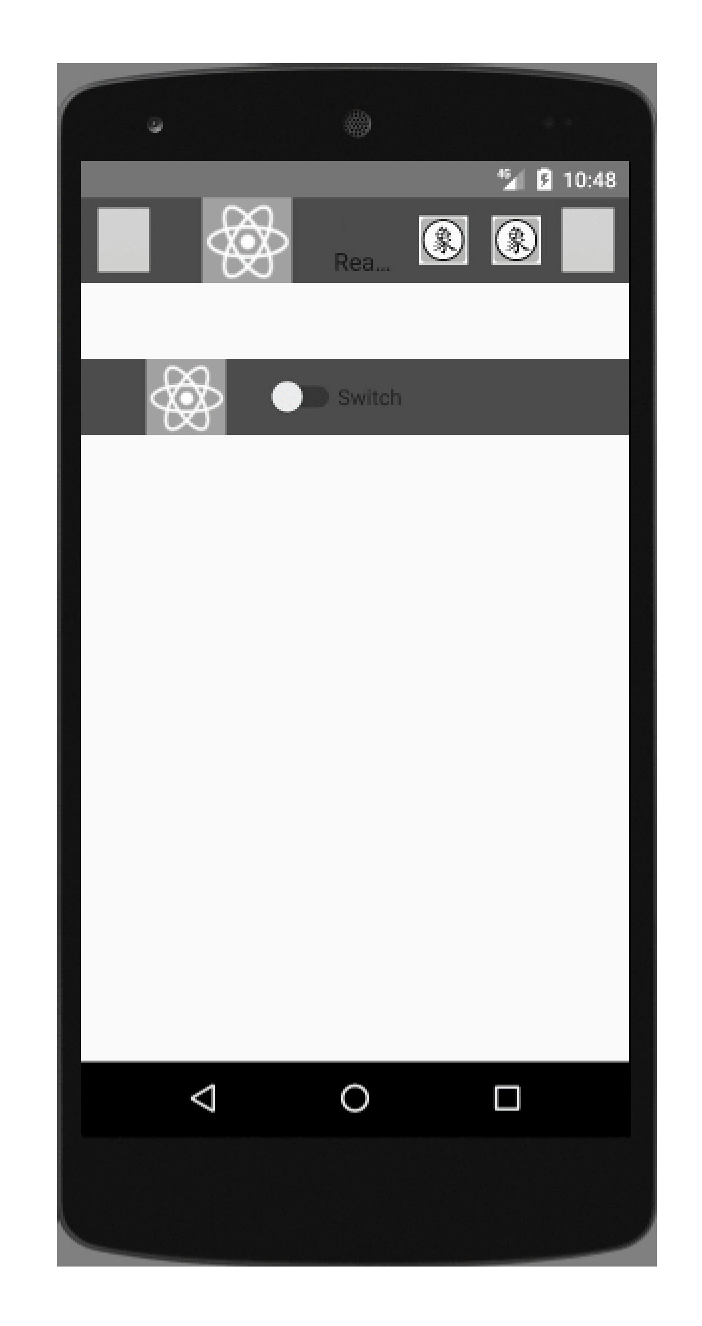


图 7-7　定制 ToolbarAndroid 组件

ToolbarAndroid 组件扩展于 View 组件，默认所有 View 组件的属性 ToolbarAndroid 组件都可以使用。ToolbarAndroid 组件常用属性如表 7-7 所示。

表 7-7　ToolbarAndroid 组件常用属性

属性名	解释	平台	值类型或可选值
navIcon	设置导航图标	Android	本地图片素材
logo	设置工具条图标	Android	本地图片素材
onIconClicked	当单击导航图片时调用的回调	Android	函数

（续表）

属性名	解释	平台	值类型或可选值

actions	设置功能按钮数组	Android	数组，其中为如下格式的对象：
			{
			title：功能按钮标题
			icon：功能按钮图标
			show：是否隐藏
			sowWithText：是否显示文案
			}
			其中 show 属性可选值如下：
			• always：总是显示
			• ifRoom：能放下就显示
			• never：不显示

onActionSelected	设置单击功能按钮后的回调函数	Android	函数，会将选中的功能按钮在数组中的下标传入
overflowIcon	设置溢出的功能按钮被集成到列表中的入口图标	Android	本地图片素材
subtitle	设置工具条副标题	Android	字符串
subtitleColor	设置工具条副标题颜色	Android	特定格式的颜色字符串
title	设置工具条标题	Android	字符串
titleColor	设置工具条标题颜色	Android	特定格式的颜色字符串

需要注意，所有 ToolbarAndroid 组件中使用到的图片素材必须为本地素材，不可以使用远程图像（虽然在开发环境可能可以使用，但正式打包时依然会有问题）。

7.6 Navigator 导航控制器

Navigator 组件是 React Native 中最为重要的几个组件之一。有了 Navigator 组件，你可以轻松地实现界面间的切换跳转。Navigator 组件并非为一个独立的视图，其实质是一种界面管理器，其管理着一组视图的切换。在学习 Navigator 组件之前，先回忆一下栈这种数据结构。

栈是编程中一种重要的数据结构，与其对应的还有队列这种数据结构。你可以将栈理解为只有一面开口的容器，数据的进出遵守“后进先出，先进后出”这种原则。队列则是一种双面开口的容器，数据的进出遵守“先进先出，后进后出”原则。栈和队列数据存储的示例图如图 7-8 所示。

通常，数据的入栈操作也叫 push，数据的出栈操作也叫 pop。在 Navigator 组件中，栈存储的其实就是每个界面。

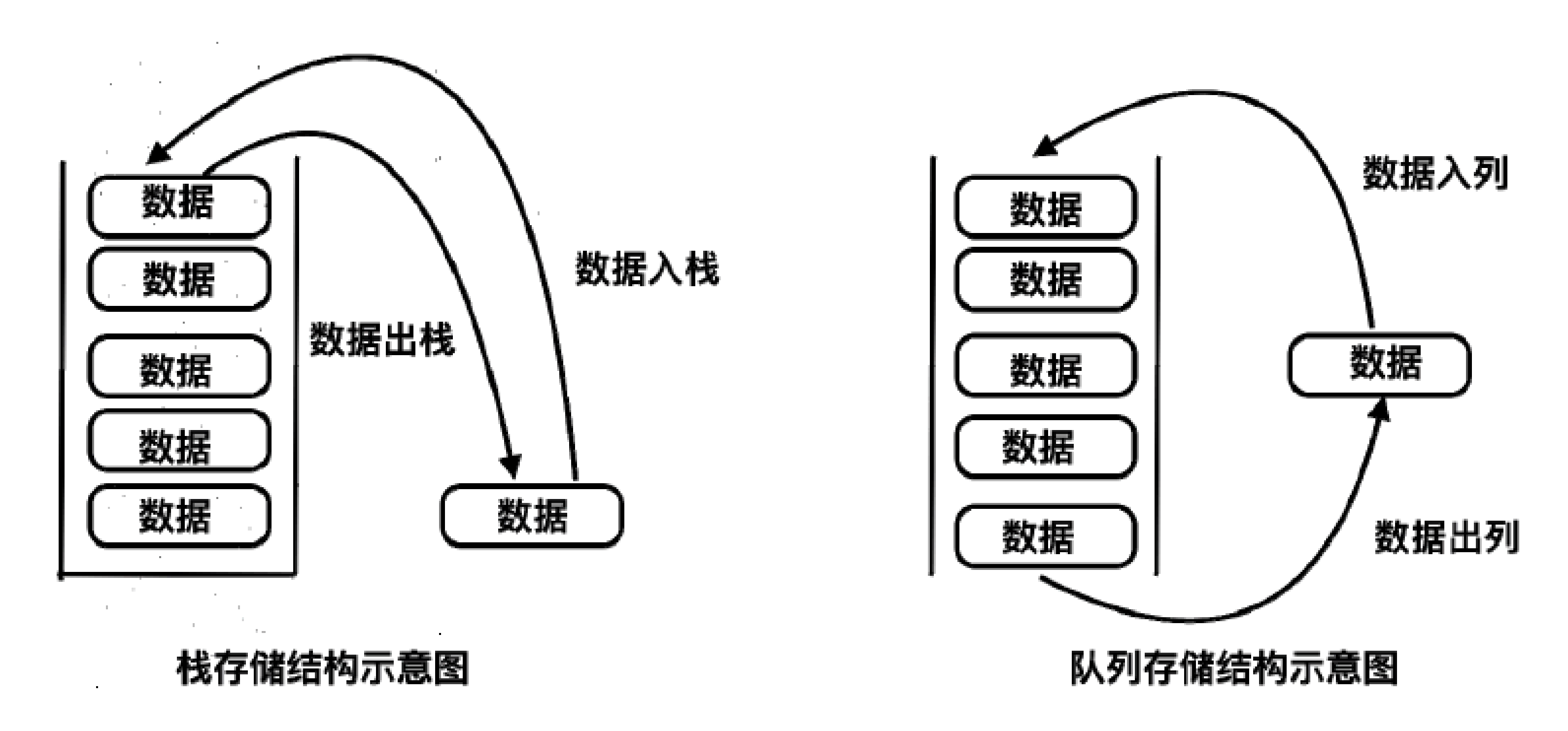


图 7-8 栈与队列存储示意图

7.6.1 Navigator 牛刀小试

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 NavigatiorDemo.js 的文件，在其中编写如下代码：

```
    import React,{Component} from 'react';
    import {Navigator,View,Text,Button} from 'react-native';
    export default class NavigatorDemo extends Component{
        render(){
            return(
                <Navigator initialRoute={{name:'first',index:0}}
                renderScene={(route,navigatior)=>{
                    return this.renderScene(route,navigatior);
                }}/>
            );
        }
        renderScene(route,navigatior){
            return(
                <View style={{flex:1,backgroundColor:'red'}}>
                    <Text style={{marginTop:30,fontSize:20,textAlign:'center'}}
>{route.name}</Text>
                    <Button title="push" onPress={()=>{
                        navigatior.push({name:'Scene'+(route.index+1),
index:route.index+1});
                    }}/>
                    <Button title="pop" onPress={()=>{
                        if (route.index>0) {
                            navigatior.pop();
                        }
                    }}/>
                </View>
            );
        }
    }
```

上面的示例代码中只使用到了 Navigator 组件的两个属性——initialRoute 属性和 renderScene 属性，initialRoute 属性设置导航组件的初始路由，renderScene 属性用来提供要渲染的界面。运行工程，单击 push 和 pop 按钮，可以观察到界面的切换效果。

路由在 Navigator 组件中是十分重要的，你需要通过路由来切换对应的界面。试想一下，打开浏览器，想要浏览一个网页时需要输入此网页的地址，一个地址对应一个网页。Navigator 组件界面的切换也是采用这样的设计思路，你可以为每个路由设置名称与下标。renderScene 方法中会传入路由信息，其需要根据路由信息来渲染指定的界面。Navigator 实例的 push 与 pop 方法分别用来界面的入栈与出栈操作，你也可以简单理解，push 方法将弹出一个新的界面覆盖在原界面上，而 pop 操作则是将当前的界面移除，将其下面的界面显示出来。

抽 丝 剥 茧

需要注意，如果 React Native 版本大于 0.44，那么上面的代码可能会无法运行，原因在于 React Native 0.44 版本后将 Navigator 组件移动到了一个单独的库中，你需要在根目录中使用如下命令进行安装：

```
yarn add react-native-deprecated-custom-components
```

并且使用如下方式进行导入：

```
import { Navigator } from 'react-native-deprecated-custom-components'
```

7.6.2 Navigator 属性配置

Navigator 组件中提供了丰富的配置属性，如表 7-8 所示。

表 7-8 Navigator 组件的配置属性

属性名	解释	平台	值类型或可选值
configureScene	配置界面弹出的效果	通用	函数，会传入如下格式参数： (route, routeStack) 其中 route 为当前跳转的路由，routeStack 为路由栈数组。需要返回如下指定值（所有值都需要加 Navigator.SceneConfigs 前缀）： • PushFromRight：从右侧压入 • FloatFromRight：右侧浮入 • FloatFromLeft：左侧浮入 • FloatFromBottom:下侧浮入 • FloatFromBottomAndroid：下侧闪入 • FadeAndroid：闪烁进入 • HorizontalSwipeJump：水平滑入 • HorizontalSwipeJumpFromRight：右侧水平滑入 • VerticalUpSwipeJump：竖直滑入 • VerticalDownSwipeJump：竖直滑入

（续有）

属性名	解释	平台	值类型或可选值
renderScene	用来渲染指定路由场景	通用	函数，会传入如下参数： (route, navigator) route 为当前跳转的路由，navigator 为导航组件实例，需要返回组件对象
initialRoute	定义启动时加载的路由	通用	自定义路由对象
initialRouteStack	提供一个初始化的路由数组	通用	数组，其中为自定义的路由对象
onWillFocus	界面切换前会被调用的回调，会将新界面的路由作为参数传入	通用	函数，会将切换界面的路由作为参数传递
onDidFocus	界面切换完成后回调的函数	通用	函数，会将当前界面的路由传入
navigationBar	设置在界面切换过程中使用保持的导航栏	通用	需要返回组件对象
navigator	设置父导航控件	通用	需要返回 Navigator 对象
sceneStyle	设置每个界面容器的风格	通用	同 View 组件的 style 属性，这个属性设置后会作用在所有路由界面中

7.6.3　Navigator 实例方法解析

Navigator 组件中提供了大量的用来转换界面的方法，如表 7-9 所示。

表 7-9　Navigator 组件的方法

方法名	解释	平台	参数
getCurretRoutes	获取当前路由栈数组	通用	无
jumpBack	跳回上一个路由，但是依然保存当前路由	通用	无
jumpForward	与 jumpBack 对应，如果通过 jumpBack 跳回上一个路由，路由会被保存，使用这个方法可以再跳回下一个路由	通用	无
jumpTo	跳转到指定路由，不销毁当前的路由	通用	路由对象
push	跳转到新的界面	通用	路由对象
pop	跳转回上一个界面，并且将当前界面和路由销毁	通用	无
replace	用一个新的路由替换当前的界面	通用	路由对象
replaceAtIndex	用新的路由替换掉路由栈数组中指定位置的路由	通用	需要两个参数，第一个为路由对象，第二个为要替换的路由在栈数组中的下标
replacePrevious	替换掉之前的路由界面	通用	路由对象
resetTo	跳转到新的场景，并且重置路由栈	通用	路由对象
immediatelyResetRouteStack	用新的路由栈来重置 Navigator	通用	路由对象数组

（续表）

方法名	解释	平台	参数
popToRoute	跳转回之前的某个路由位置，这个位置之后的所有路由界面都会销毁	通用	路由对象
popToTop	跳转回导航的第一个界面，之后所有的路由界面都会被销毁	通用	无

7.7 iOS 平台的导航控制器 NavigatorIOS 组件

如果你熟悉 iOS 原生开发，那么你一定知道 UINavigationController 这个类，在 iOS 原生开发中会时常使用到 UINavagationController 来组织导航栈结构的界面切换。React Native 中提供了与其对应的 NavigatorIOS 组件。

7.7.1 使用 NavigatorIOS 组件

在 HelloWorld 工程的 Demo 文件夹下面新建一个命名为 NavigatorIOSDemo.js 的文件，在其中编写如下代码：

```
import React, {
    Component
} from 'react';
import {
    NavigatorIOS,
    View,
    Text,
    Button
} from 'react-native';
export default class NavigatorIOSDemo extends Component {
    render() {
        return (
            <NavigatorIOS initialRoute = {
                {
                    component: MyScene,
                    title: 'First',
                    passProps:{
                        param:"Secen",
                        index:0
                    }
                }
            }
            style = {
                {
                    flex: 1
```

```
                }
            }
            />
        );
    }
}
class MyScene extends Component {
    render() {
        return (
            <View>
                <Text style={{marginTop:70,textAlign:'center',fontSize:22}}>
                    {this.props.param+this.props.index}
                </Text>
                <Button title = "push"
                    onPress = {
                        () => {
                            this.props.navigator.push({
                            title: "Other",
                            component:MyScene,
                            passProps:{
                                param:"Secen",
                                index:this.props.index+1
                            }
                        });
                    }
                }
            />
            </View> )
        ;
    }
}
```

修改 index.ios.js 文件后在 iOS 平台运行工程，效果如图 7-9 所示。

NavigatorIOS 组件自带一个顶部的导航栏，导航栏中间会显示标题，如果当前界面不是导航器的根界面，那么导航栏上还会默认显示一个返回按钮，当用户单击返回按钮或者使用右滑手势时，可以实现界面的返回操作（可以单击 push 进行试验）。

关于上面的示例代码，我们还需要深入地解释一下，和 Navigator 组件不同的是，NavigatorIOS 组件的路由配置有着严格的要求，同时功能也更加强大。首先 NavigatorIOS 组件的 initialRoute 属性用来设置导航器的初始路由，即导航加载出来时默认显示的界面。在 NavigatorIOS 组件中，路由对象是被严格定义的，其中可以添加的属性和意义如表 7-10 所示。

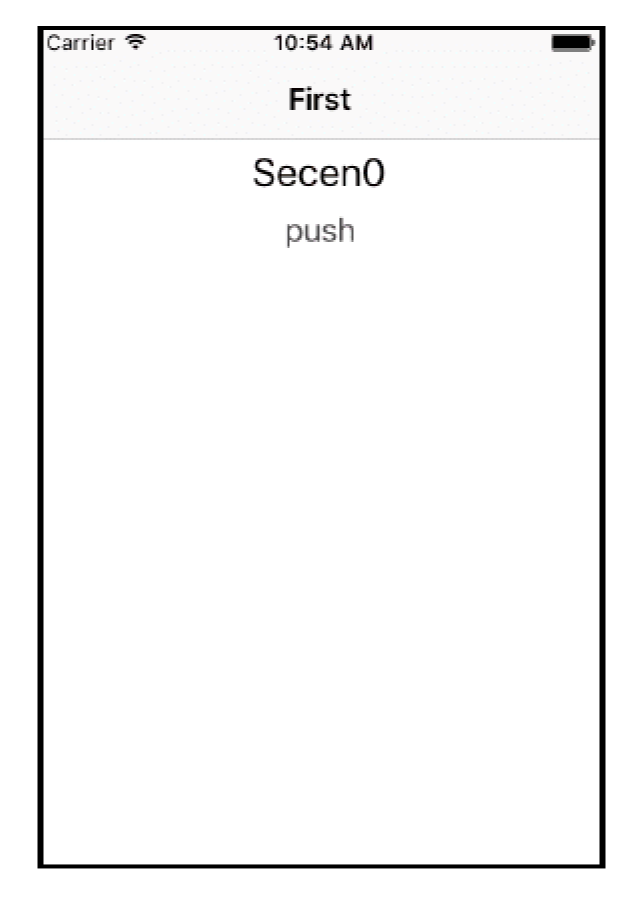


图 7-9　NavigatorIOS 组件效果

表 7-10　在 NavigatorIOS 组件路由对象属性

属性名	解释	平台	值类型或可选值
component	必选属性，设置渲染界面的组件类	iOS	类函数对象
title	设置界面标题，这个标题会被渲染在导航栏中间	iOS	字符串
titleImage	设置标题图标，如果设置了这个属性，会在导航栏中间显示所设置的图标	iOS	本地图片素材
passProps	用来传递参数	iOS	自定义对象
backButtonIcon	设置返回按钮图标	iOS	本地图片素材
backButtonTitle	设置返回按钮标题	iOS	字符串
leftButtonTitle	提供一个左侧功能按钮，设置按钮标题	iOS	字符串
leftButtonIcon	提供一个左侧功能按钮，设置按钮图标	iOS	本地图片素材
leftButtonSystemIcon	提供一个系统风格的左侧功能按钮	iOS	枚举字符串： • done：完成按钮 • cancel：取消按钮 • edit：编辑按钮 • save：存储按钮 • add：添加按钮 • compose：整理按钮 • reply：重复按钮 • action：活动按钮 • organize：组织按钮 • bookmarks：书库按钮 • search：搜索按钮 • refresh：刷新按钮 • stop：停止按钮 • camera：相机按钮 • trash：删除按钮 • play：播放按钮 • paush：暂停按钮 • rewind：回退按钮 • undo：撤销按钮 • redo：重做按钮
onLeftButtonPress	左侧功能按钮被单击时的回调函数	iOS	函数
rightButtonIcon	设置右侧功能按钮图标	iOS	本地图片素材
rightButtonTitle	设置右侧功能按钮标题	iOS	字符串
rightButtonSystemIcon	提供一个系统风格的右侧功能按钮	iOS	枚举字符串，同 rleftButtonSystemIcon
onRightButtonPress	右侧功能按钮被单击时回调的函数	iOS	函数

（续表）

属性名	解释	平台	值类型或可选值
wrapperStyle	设置当前路由对应界面中容器视图的风格	iOS	style 对象
navigationBarHidden	设置是否隐藏导航栏	iOS	布尔值
shadowHidden	设置是否隐藏导航栏下边线	iOS	布尔值
tintColor	设置导航栏按钮颜色	iOS	颜色字符串
barTintColor	设置导航栏背景颜色	iOS	颜色字符串
titleTextColor	设置导航栏标题颜色	iOS	颜色字符串
translucent	设置导航栏是否半透明	iOS	布尔值

正如表 7-10 所示，NavigatorIOS 组件的路由对象提供了极为丰富的可配置属性，当进行界面渲染时，component 属性注册的组件类会被自动进行实例化，并且会对界面组件实例添加一些属性：route、navigator 以及 passProps 所提供的自定义对象中的属性。在 MyScene 类中，你可以使用 this.props.navigator 来获取导航实例，也可以使用 this.props.route 获取当前路由对象，还可以使用 this.props.index 直接获取 passProps 传递的属性。

7.7.2　NavigatorIOS 属性与方法解析

NavigatorIOS 组件中常用属性如表 7-11 所示。

表 7-11　NavigatorIOS 组件常用属性

属性名	解释	平台	值类型或可选值
navigationBarHidden	设置导航栏是否隐藏	iOS	布尔值
shadowHidden	设置是否显示导航栏下边线	iOS	布尔值
itemWrapperStyle	设置导航中界面组件的容器视图风格	iOS	style 对象
tintColor	设置导航按钮颜色	iOS	颜色字符串
barTintColor	设置导航栏背景色	iOS	颜色字符串
titleTextColor	设置导航标题颜色	iOS	颜色字符串
translucent	设置导航栏是否半透明	iOS	布尔值
interactivePopGestureEnabled	设置是否开启右滑自动返回上一界面手势	iOS	布尔值
initialRoute	初始路由	iOS	路由对象

NavigatorIOS 组件提供的界面切换方法与 Navigator 组件基本一致，只是传入的路由参数必须是严格的 NavigatorIOS 路由对象，方法如表 7-12 所示。

表 7-12　NavigatorIOS 路由对象方法

方法名	解释	平台	参数
push	切换到一个新的界面	iOS	路由对象
pop	返回上一个界面	iOS	无
popN	返回到前 N 个界面	iOS	数值，当传入 1 时效果和 pop 一致

（续表）

方法名	解释	平台	参数
replaceAtIndex	替换导航栈中某个路由	iOS	传参格式如下： (route,index) • route：路由对象 • index：替换导航栈中的路由下标
replace	替换当前界面的路由	iOS	路由对象
replacePrevious	替换上一个界面的路由	iOS	路由对象
popToTop	返回到导航栈根路由界面	iOS	无
popToRoute	返回到导航栈中的指定路由	iOS	路由对象
replacePreviousAndPop	替换上一个界面的路由并且执行 pop 操作	iOS	路由对象
resetTo	重置导航栈	iOS	路由对象

抽 丝 剥 茧

NavigatorIOS 组件中最重要的概念便是路由，路由配置负责对界面渲染、设置导航栏样式以及为数据传递提供支持。

7.8 标签栏 TabBarIOS 组件

本节将介绍 React Native 中最后一个常用的视图管理组件 TabBarIOS。正如其名，TabBarIOS 组件只能应用于 iOS 平台，其是对 iOS 原生控件 UITabBarBarController 的 React Native 版实现。所谓标签栏其实际也是用于界面的切换，与 Navigator 不同的是导航管理的界面是有层次结构的，而标签栏管理的界面是并列结构，其界面的切换并没有父子关系。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 TabBarIOSDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {TabBarIOS,View,Text} from 'react-native';
export default class TabBarIOSDemo extends Component{
    constructor(props){
        super(props);
        this.state={
            item1Selected:true,
            item2Selected:false
        }
    }
    render(){
        return(
            <TabBarIOS
```

```
                barTintColor="green"
                style={{height:49}}
                tintColor="red"
                unselectedItemTintColor="white"
                translucent={false}>
                    <TabBarIOS.Item
                    title="历史"
                    systemIcon="history"
                    selected={this.state.item1Selected}
                    badge={1}
                    onPress={()=>{
                        this.setState({
                            item1Selected:true,
                            item2Selected:false
                            });
                    }}>
                        <View>
                            <Text style={{top:100,textAlign:'center',
fontSize:30}}>历史界面</Text>
                        </View>
                    </TabBarIOS.Item>
                    <TabBarIOS.Item
                    title="数学"
                    systemIcon="bookmarks"
                    selected={this.state.item2Selected}
                    onPress={()=>{
                        this.setState({
                            item1Selected:false,
                            item2Selected:true
                            });
                    }}>
                        <View>
                            <Text style={{top:100,textAlign:'center',
fontSize:30}}>数学界面</Text>
                        </View>
                    </TabBarIOS.Item>
                </TabBarIOS>
            );
        }
    }
```

TabBarIOS 组件中的属性用来配置导航栏，是一个双标签，其中需要嵌套 TabBarIOS.Item 组件。Item 组件用来配置标签栏中每个具体的标签，也是一个双标签，在 Item 组件中进行对应界面视图的编写。在 iOS 平台运行工程，效果如图 7-10 所示。

TabBarIOS 组件十分简洁，扩展于 View 组件，默认可以使用 View 组件中包含的属性，其他用于配置标签栏的属性如表 7-13 所示。



图 7-10　TabBarIOS 组件效果

表 7-13　TabBarIOS 组件属性

属性名	解释	平台	值类型或可选值
unselectedTintColor	设置未选中状态下的标签 Item 文字颜色	iOS	颜色字符串
tintColor	设置选中状态下的标签 Item 图标颜色	iOS	颜色字符串
unselectedItemTintColor	设置未选中状态下的标签 Item 图标颜色	iOS	颜色字符串
barTintColor	设置标签栏背景色	iOS	颜色字符串
translucent	设置标签栏是否半透明	iOS	布尔值
itemPositioning	设置标签栏中标签 Item 图标显示模式	iOS	枚举字符串 • fill：充满 • center：居中 • auto：自动

TabBarIOS.Item 实际上是 TabBarIOSItem 组件，其中提供的常用属性如表 7-14 所示。

表 7-14　TabBarIOS.Item 的常用属性

属性名	解释	平台	值类型或可选值
badge	设置此标签头标气泡显示的文案	iOS	字符串或数值
badgeColor	设置此标签头标气泡的颜色	iOS	颜色字符串
systemIcon	设置此标签显示为系统图标	iOS	枚举字符串 • bookmarks：书库图标 • contacts：联系人图标 • downloads：下载图标 • favorites：偏好图片 • history：历史图标 • more：更多图标 • most-recent：最近图标

（续表）

属性名	解释	平台	值类型或可选值
systemIcon	设置此标签显示为系统图标	iOS	• most-viewed：浏览图标 • recents：最近图标 • search：搜索图标 • top-rated：排行图标
icon	为此标签设置一个自定义的图标	iOS	本地图片素材
selectedIcon	为此标签设置一个选中状态的图标	iOS	本地图片素材
onPress	当单击标签时触发的回调	iOS	函数
renderAsOriginal	图标的渲染是否保持原始状态	iOS	布尔值
selected	设置此标签是否选中	iOS	布尔值
style	设置标签的风格属性	iOS	style 对象
title	设置标签的标题	iOS	字符串

需要注意，TabBarIOSItem 组件是一个受控组件，也就是说除非手动修改 selected 属性的值，否则标签的选中状态不会随用户的操作进行修改。通常情况下，开发者会通过属性来确定标签栏的选中状态，当 onPress 方法被调用时来进行属性的刷新。

博 闻 强 识
在 iOS 原生开发中，一种常用的界面搭建框架便是在标签栏中嵌套导航，标签栏并列展示几个功能模块，导航来控制控能模块中子级界面的切换。

第 8 章

React Native 技能进阶

通过前几章的学习，你已经学会了使用 React Native 中大部分组件编写一些小的有趣的示例程序。看到自己的学习成果在 iOS 或 Android 设备上成功运行时，你一定体会到了编程的喜悦。但是，要掌握开发完整商业应用的能力，仅仅学会使用这些独立组件是远远不够的，你需要将这些独立组件进行适当的组合与嵌套才能构建完整的界面，将界面再通过导航器或标签器适当地管理才能构成完整的应用程序。你还需要学习使用网络技术来获取互联网上的数据供自己的应用程序所用，使用存储技术对下载的数据进行持久化保存等。所以，你的 React Native 学习之旅任重而道远，继续加油吧！

有关 React Native 中的界面布局技术，虽然在前面的学习中，有意或无意地使用到了一些简单的布局属性，但是和 React Native 完整的布局模型比起来，你体验到的只是冰山一角，通过本章的学习，你将能够根据设计图完成复杂而精准的布局。本章还会介绍 React Native 中的动画技术，移动应用的出彩之处便在于其支持强大的动画特效，恰当地使用动画，可以极大地提高用户体验。网络和数据存储技术在开发完整应用程序中也是必不可少的。除此之外，本章还将介绍 React Native 中的更多进阶技巧，例如警告弹框、键盘、通知等技术。

8.1　React Native 布局技术

有界面就有布局，界面不可能脱离布局独立存在。React Native 中界面的布局实质上就是组件的嵌套与组合，你可以使用 Flexbox 思想来对 React Native 中的组件做精准的定位。

博 闻 强 识
Flexbox 是 Web 开发中的一种布局模式，其常常被叫作弹性盒布局模式。如今 Web 界面的开发多采用 div+css 模式，其实际上就是这种弹性盒布局。

8.1.1 布局中的主轴与次轴

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 LayoutDemo.js 的文件，在其中编写如下代码：

```
    import React,{Component} from 'react';
    import {View} from 'react-native';
    export class LayoutDemoOne extends Component{
        render(){
            return(
                <View>
                    <View style={{backgroundColor:'red',height:30,
width:30}}></View>
                    <View style={{backgroundColor:'green',height:30,
width:30}}></View>
                    <View style={{backgroundColor:'blue',height:30,
width:30}}></View>
                    <View style={{backgroundColor:'black',height:30,
width:30}}></View>
                </View>
            );
        }
    }
```

需要注意，这个例子和我们之前写的例子有略微的不同，其并没有设置导出的类为默认类，这是因为本节你将在同一个文件中编写多个布局测试类。修改 index.ios.js 与 index.android.js 文件后运行工程，可以看到效果如图 8-1 所示，4 个 View 组件依次垂直排列。

抽 丝 剥 茧

导出的类如果没有使用 default 修饰为默认导出类，则在 index 文件中导入时需要使用如下方式:

```
    import {LayoutDemoOne} from './Demo/LayoutDemo';
```

从上面的示例可以看出，默认的视图排列方式是竖直排列的，每个子组件占其父组件内的一行空间。其实在 React Native 布局结构中，父容器组件可以通过设置相应的布局属性来设置其中子组件的主轴布局方向。所谓主轴布局方向，是指此组件内部的组件以何种流式方向进行布局，主要分为水平和竖直两种，当然水平布局又分为从左向右与从右向左，竖直布局又分为从上到下与从下到上。通过父容器组件的 flexDirection 属性来进行子组件的布局方向设置，其可以设置的值有 4 种，如表 8-1 所示。

表8-1 flexDirection属性值

属性值	说明
row	从左向右水平布局
row-reverse	从右向左水平布局
column	从上到下竖直布局
column-reverse	从下到上竖直布局

例如，将上面示例代码的主轴布局方向修改为 row，代码如下，效果如图 8-2 所示。

```
<View style={{flexDirection:'row'}}>
    <View style={{backgroundColor:'red',height:30,width:30}}></View>
    <View style={{backgroundColor:'green',height:30,width:30}}></View>
    <View style={{backgroundColor:'blue',height:30,width:30}}></View>
    <View style={{backgroundColor:'black',height:30,width:30}}></View>
</View>
```

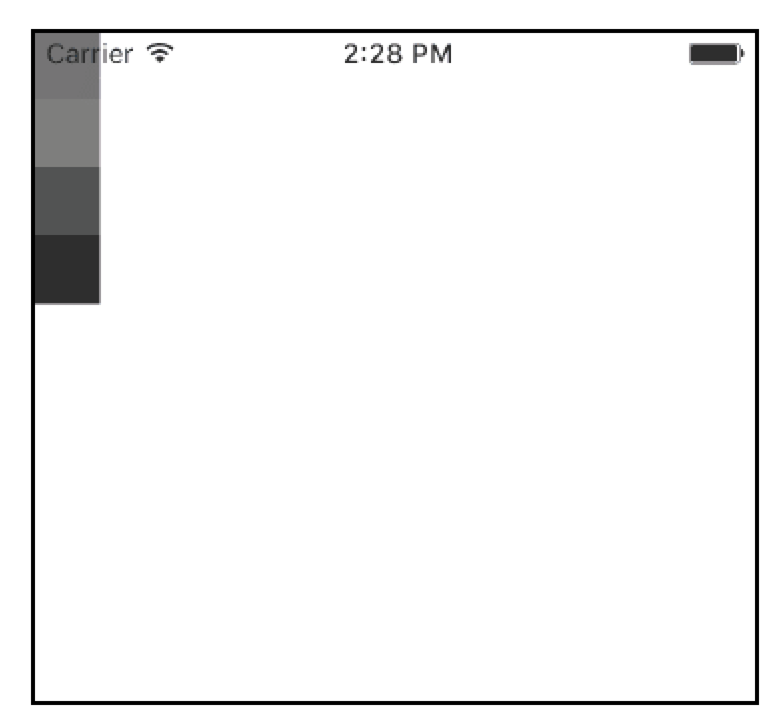


图 8-1 默认的布局排列方式

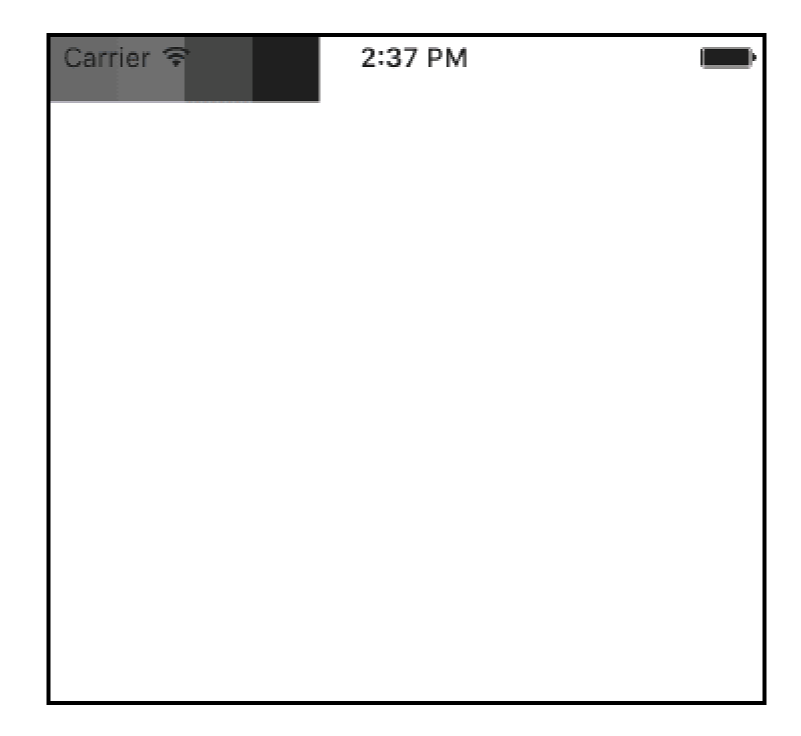


图 8-2 进行水平布局

从上面的布局效果可以看出，各个子组件是从父组件的边缘紧密连续布局的，可以通过设置父容器组件的 justifyContent 属性来设置子组件的布局模式，可选值如表 8-2 所示。

表8-2 justifyContent属性值

属性值	说明
flex-start	从父容器起始端开始布局
flex-end	从父容器结束端开始布局
center	从父容器中心开始布局
space-around	预留边距并且子组件间也等间距布局
space-between	不留边距并且子组件间等间距布局

修改代码如下，再次运行，效果如图 8-3 所示。

```
<View style={{flexDirection:'row',justifyContent:"space-around"}}>
    <View style={{backgroundColor:'red',height:30,width:30}}></View>
    <View style={{backgroundColor:'green',height:30,width:30}}></View>
```

```
    <View style={{backgroundColor:'blue',height:30,width:30}}></View>
    <View style={{backgroundColor:'black',height:30,width:30}}></View>
</View>
```

在布局的过程中，默认父容器组件的尺寸会根据其内部子组件所占空间进行调整（这也是 Flexbox 布局思想的核心），也可以手动指定父容器组件的尺寸，如果指定父组件的尺寸要比其内子组件所占尺寸大得多，就可能需要使用到父容器组件的 alignItems 属性，这个属性用来设置子组件在次轴方向的布局方式。次轴是指与主轴所垂直的轴，例如，如果主轴为水平方向，则次轴就是竖直方向，如果主轴是竖直方向，那么次轴就是水平方向。alignItems 可设置的值如表 8-3 所示。

表8-3　alignItems属性值

属性值	说明
flex-start	从次轴起始端开始布局
flex-end	从次轴结束端开始布局
center	从次轴中间开始布局
stretch	子组件在次轴方向上的尺寸充满容器，这个属性要想生效，子组件不可以有对应方向上的尺寸值设置

修改布局代码如下，再次运行工程，效果如图 8-4 所示。

```
<View style={{flexDirection:'row',
             justifyContent:"space-around",
             height:300,
             alignItems:'center',
             backgroundColor:'yellow'
            }}>
    <View style={{backgroundColor:'red',height:30,width:30}}></View>
    <View style={{backgroundColor:'green',height:30,width:30}}></View>
    <View style={{backgroundColor:'blue',height:30,width:30}}></View>
    <View style={{backgroundColor:'black',height:30,width:30}}></View>
</View>
```

如果需要各个子组件在次轴上有不同的布局模式，可以直接设置子组件的 alignSelf 属性，可选值如表 8-4 所示。

表8-4　alignSelf属性值

属性值	说明
auto	自动继承父容器的 alignSelf 的值
felx-start	从父容器次轴起始端开始布局
flex-end	从父容器次轴结束端开始布局
center	从父容器次轴中心开始布局
stretch	充满父容器次轴

修改布局代码如下，运行工程，效果如图 8-5 所示。

```
    <View style={{flexDirection:'row',
                 justifyContent:"space-around",
                 height:300,
                 alignItems:'center',
                 backgroundColor:'yellow'
                }}>
        <View style={{backgroundColor:'red',alignSelf:'stretch',
width:30}}></View>
        <View style={{backgroundColor:'green',alignSelf:'flex-start',height:30,
width:30}}></View>
        <View style={{backgroundColor:'blue',alignSelf:'flex-end',height:30,
width:30}}></View>
        <View style={{backgroundColor:'black',alignSelf:'center',height:30,
width:30}}></View>
    </View>
```

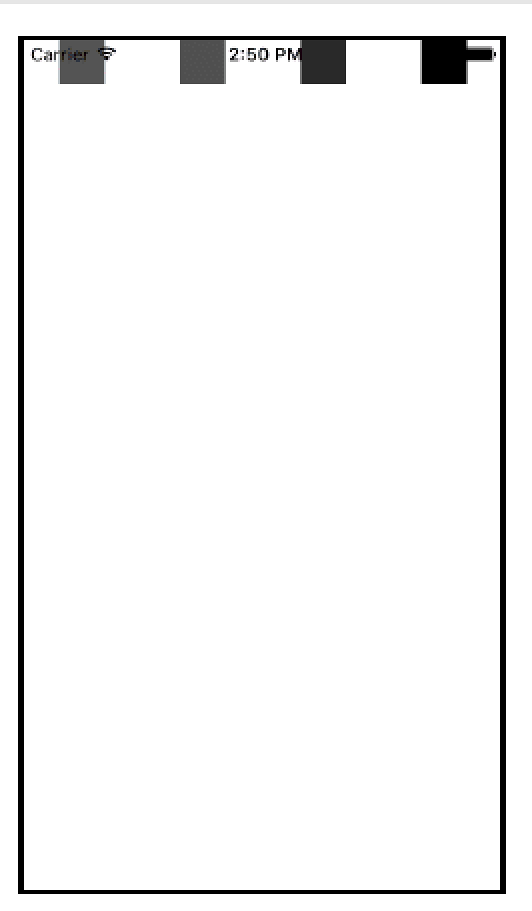


图 8-3　等间距进行布局

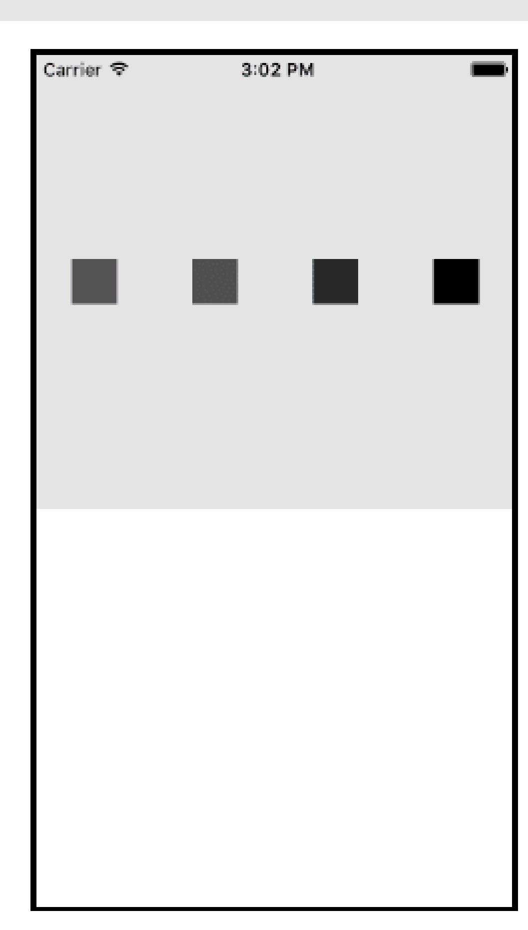


图 8-4　在次轴上进行居中布局

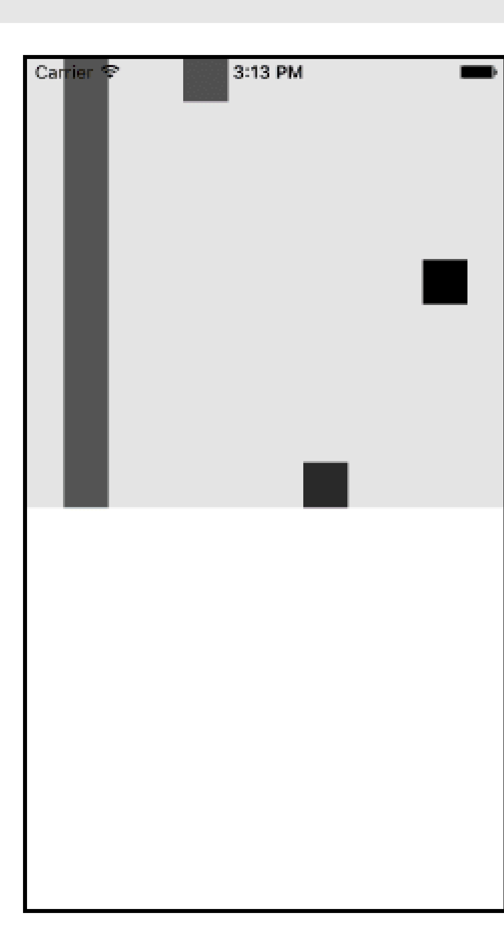


图 8-5　灵活设置子组件的次轴布局模式

理解与掌握了本小节中介绍的 Flexbox 布局思想，基本就可以在 React Native 中完成大部分界面布局效果了。如果一个界面既有水平布局的元素也有竖直布局的元素该怎么办呢？其实很简单，合理地进行组件的嵌套即可。例如，可以在竖直布局的 View 中嵌套几个水平布局的 View，也可以在水平布局的 View 中继续嵌套竖直布局的 View，总之同样的界面布局的方式多种多样，正如一句谚语所说“条条大路通罗马”。

8.1.2　精准定义组件的尺寸

通过上一小节的学习，应该学会了如何通过主轴与次轴进行简单的组件排布。确定组件的位置只是界面布局中的一个部分，更重要的是可以灵活地控制组件的尺寸。在 React Native 中，组件的尺寸可以直接通过设置 width 和 height 属性进行确定，这也是前面示例代码我们常采用的方式，在 LayoutDemo.js 文件中新创建一个类，代码如下：

```
    export class LayoutDemoTwo extends Component{
        render(){
```

```
            return(
                <View style={{height:300,flexWrap:'nowrap',backgroundColor:
'yellow',overflow:'visible'}}>
                    <View style={{backgroundColor:'red',width:30, height:30}}>
</View>
                    <View style={{backgroundColor:'green',width:30, height:60}}>
</View>
                    <View style={{backgroundColor:'blue',width:80, height:30}}>
</View>
                    <View style={{backgroundColor:'black',width:30,
height:100}}></View>
                </View>
            );
        }
    }
```

运行工程，效果如图 8-6 所示。

上面示例代码中的子组件高度和并没有超出父组件，如果组件的高度和超出了父组件高度，就可以通过以下两种方式来处理。

1. 设置超出部分被截取或者超出部分依然显示

子组件超出部分的显隐可以通过 overflow 属性来进行设置，如果设置为 visiable，则超出部分依然可以显示，如果设置为 hidden，则超出部分会被隐藏，示例代码如下：

```
    <View style={{height:300,flexWrap:'nowrap',backgroundColor:'yellow',
overflow:'visible'}}>
        <View style={{backgroundColor:'red',width:30,height:30}}></View>
        <View style={{backgroundColor:'green',width:30,height:60}}></View>
        <View style={{backgroundColor:'blue',width:80,height:30}}></View>
        <View style={{backgroundColor:'black',width:30,height:300}}></View>
    </View>
```

运行工程，效果如图 8-7 所示。

2. 通过设置自动折行来排布子组件

设置父组件的 flexWrap 属性可以实现超出父容器组件的子组件折行显示，这个属性设置为 warp 则表示要支持折行，设置为 nowarp 表示不支持折行，示例如下：

```
    <View style={{height:300,flexWrap:'nowrap',backgroundColor:'yellow',
overflow:'visible',flexWrap:'wrap'}}>
        <View style={{backgroundColor:'red',width:30,height:30}}></View>
        <View style={{backgroundColor:'green',width:30,height:60}}></View>
        <View style={{backgroundColor:'blue',width:80,height:30}}></View>
        <View style={{backgroundColor:'black',width:30,height:300}}></View>
    </View>
```

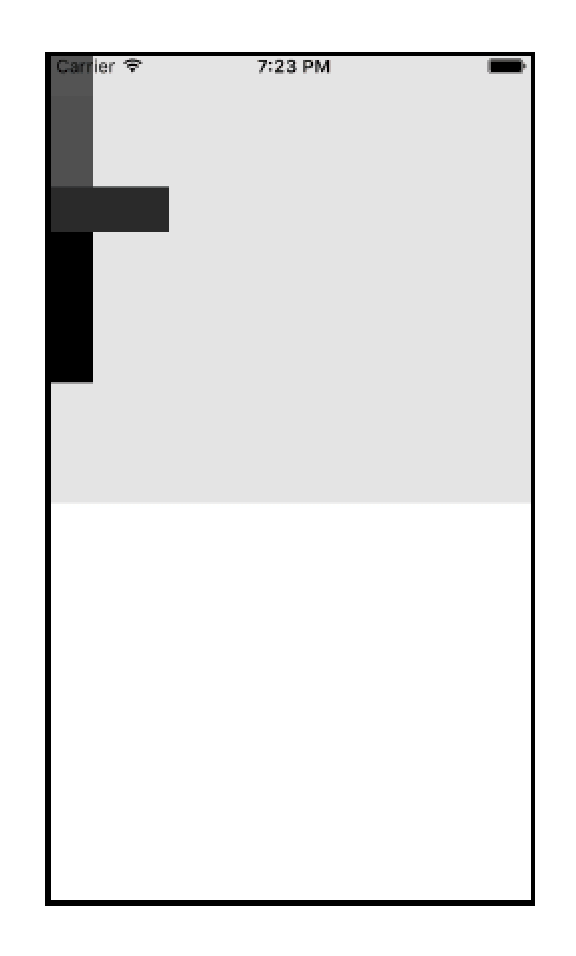



图 8-6　精确定义控件的尺寸

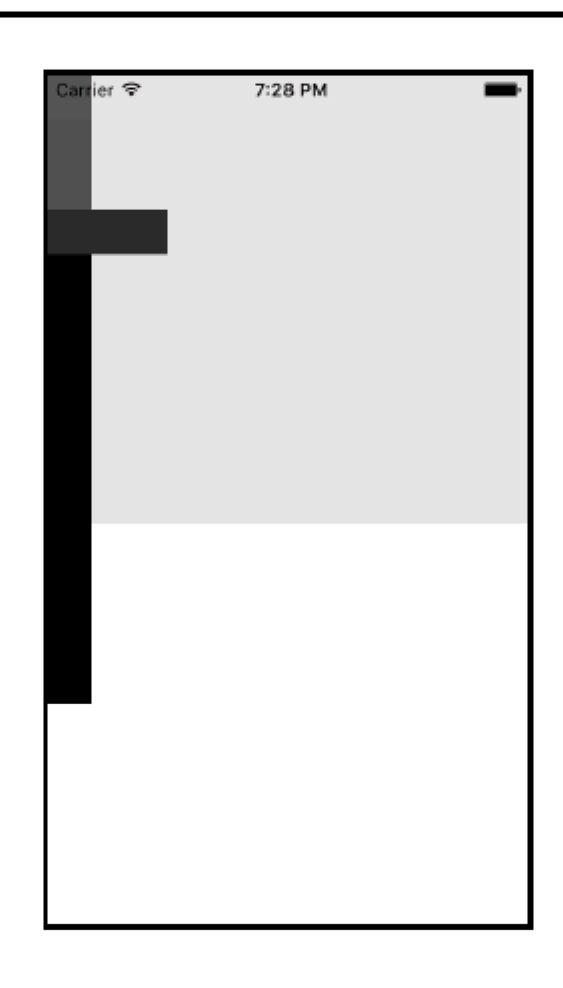


图 8-7　超出父组件的部分依然显示

运行效果如图 8-8 所示。

通过 width 和 height 属性可以达到精准定义组件尺寸的目的，但是在实际开发中，通常并不会将组件的尺寸定义死，尤其是在移动设备上，屏幕的尺寸众多，相同的设备还有横竖屏之分，因此开发者通常采用权重值的方式来定义组件的尺寸。flex 属性用来设置组件的权重值，权重相同的组件在主轴方向占据父容器中相同的宽度，示例代码如下：

```
    <View style={{height:300,flexWrap:'nowrap',backgroundColor:'yellow',
overflow:'visible',flexWrap:'wrap'}}>
        <View style={{backgroundColor:'red',width:30,flex:1}}></View>
        <View style={{backgroundColor:'green',width:30,flex:1}}></View>
        <View style={{backgroundColor:'blue',width:80,flex:2}}></View>
        <View style={{backgroundColor:'black',width:30,flex:4}}></View>
    </View>
```

上面代码设置的第 1 个子组件与第 2 个子组件占据相同的宽度，第 3 个子组件宽度是前两个的两倍，第 4 个子组件的宽度是第 3 个子组件的两倍，效果如图 8-9 所示。

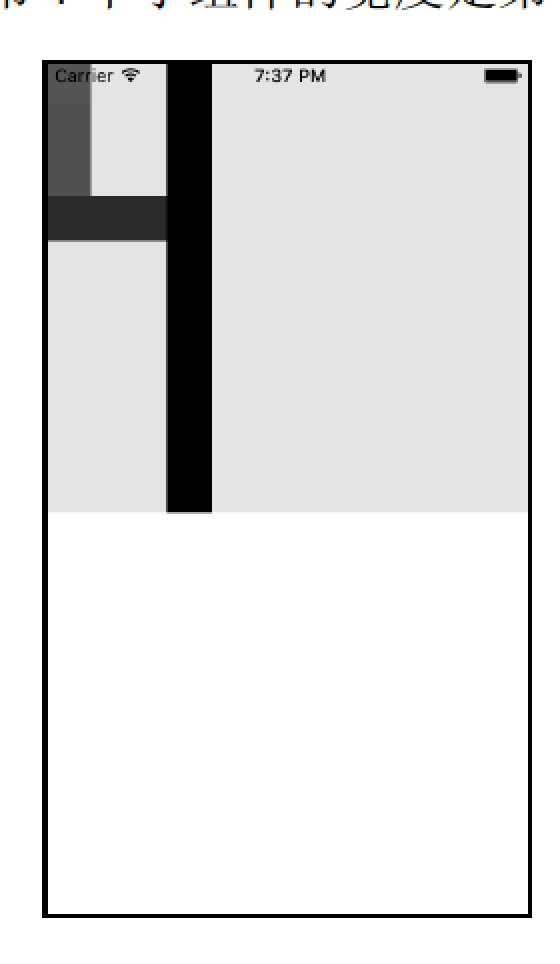


图 8-8　子组件自动折行

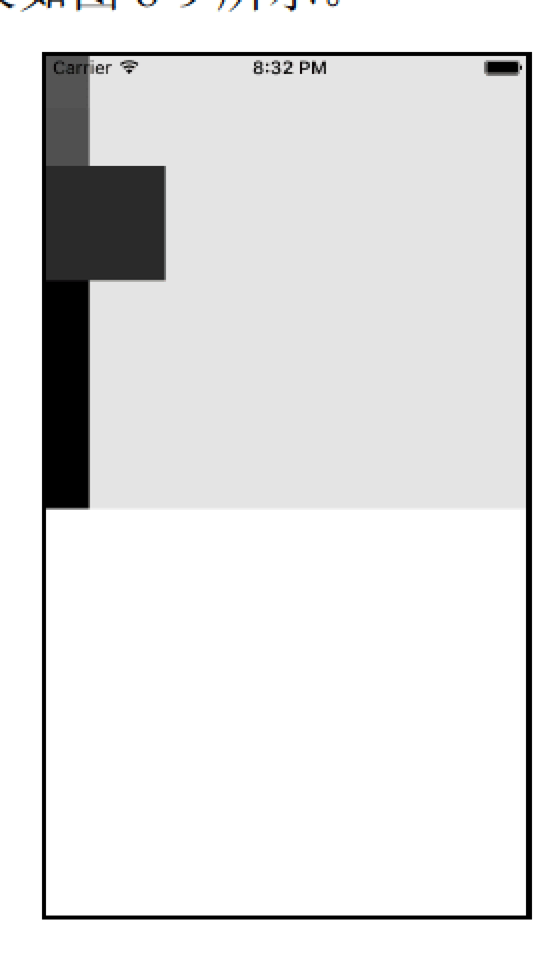


图 8-9　通过权重确定子组件的尺寸

React Native 中还提供了 maxHeight、maxWidth、minHeight、minWidth 几个属性，这个 4 个属性分别用来设置组件的最大高度、最大宽度、最小高度和最小宽度。通过这几个最值与权重的结合使用，可以使 React Native 的布局结构更加灵活。

8.1.3 相对定位与绝对定位

React Native 在进行组件布局定位时分为相对定位与绝对定位，默认情况下都是相对定位的。所谓相对定位，是指相对于组件本来应在的位置进行偏移，例如可以使用 top、bottom、left、right 属性对组件的位置进行微调，在 LayoutDemo.js 文件中新建一个类，示例代码如下：

```
    export class LayoutDemoThree extends Component{
        render(){
            return(
                <View style={{height:300,backgroundColor:'yellow',
alignItems:'center'}}>
                    <View style={{backgroundColor:'red',width:30,height:30,
bottom:10}}></View>
                    <View style={{backgroundColor:'green',width:30,height:30,
left:30}}></View>
                    <View style={{backgroundColor:'blue',width:80,
height:30}}></View>
                    <View style={{backgroundColor:'black',width:30,height:30,
right:10,top:30}}></View>
                </View>
            );
        }
    }
```

运行工程，效果如图 8-10 所示。

top、bottom、left、right 这 4 个属性只是针对原定位位置的调整，并不会影响到周围组件的布局位置。通过设置 position 属性可以修改定位模式，设置为 absolute 则表示绝对布局，设置为 relative 则表示为相对布局。默认为 relative，如果设置为 absolute 则上面 4 个属性是相对于父容器进行定位，示例如下：

```
    export class LayoutDemoThree extends Component{
        render(){
            return(
                <View style={{height:300,backgroundColor:'yellow'}}>
                    <View style={{backgroundColor:'red',width:30,height:30,
position:'absolute',top:30}}></View>
                    <View style={{backgroundColor:'green',width:30,height:30,
position:'absolute',top:100,left:100}}></View>
                    <View style={{backgroundColor:'blue',width:80,height:30,
position:'absolute',top:30,left:100}}></View>
                    <View style={{backgroundColor:'black',width:30,height:30,
position:'absolute',bottom:50}}></View>
```

```
                </View>
            );
        }
    }
```

运行工程，效果如图 8-11 所示。

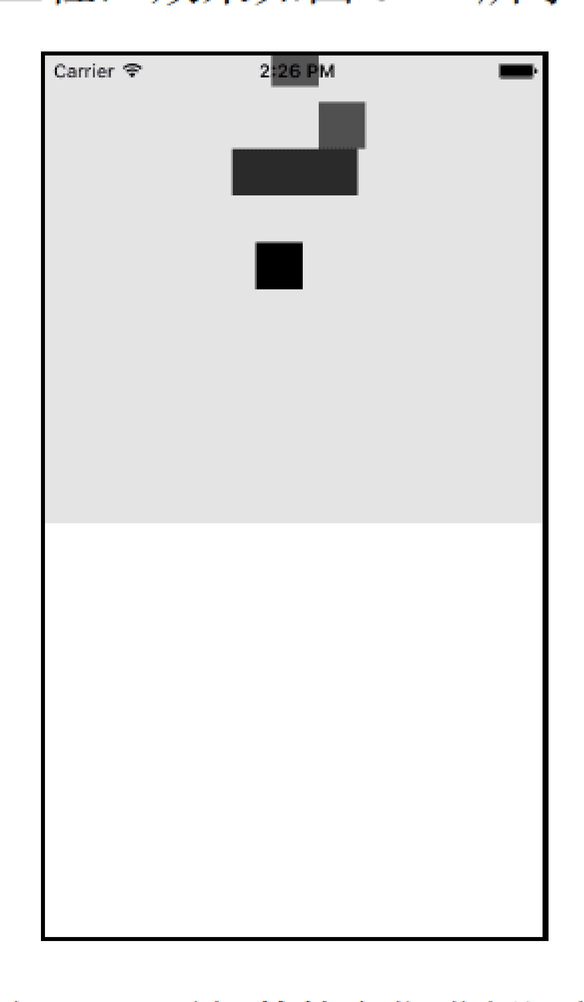



图 8-10　对组件的定位进行调整

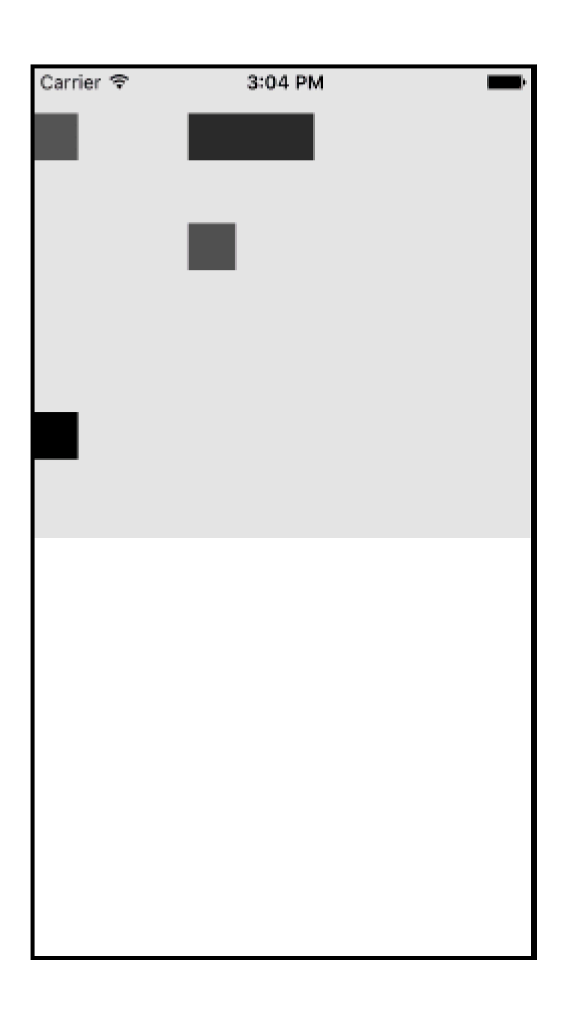


图 8-11　对组件进行绝对定位

React Native 中还提供了与 margin 相关的属性，可以用来设置组件的外间距。margin 相关属性如表 8-5 所示。

表 8-5　margin 的相关属性

属性	说明
marginTop	设置组件上外边距
marginBottom	设置组件下外边距
marginLeft	设置组件左外边距
marginRight	设置组件右外边距
marginHorizontal	设置组件水平边距（相当于同时设置 marginLeft 与 marginRight）
marginVertical	设置组件竖直边距（相当于同时设置 marginTop 与 marginBottom）
margin	设置组件外边距（相当于同时设置 marginTop、marginBottom、marginLeft 与 marginRight）

示例代码如下：

```
    export class LayoutDemoThree extends Component{
        render(){
            return(
                <View style={{height:300,backgroundColor:'yellow'}}>
                    <View style={{backgroundColor:'red',width:30,height:30,
marginTop:30}}></View>
                    <View style={{backgroundColor:'green',width:30,height:30,
marginLeft:30}}></View>
```

```
                    <View style={{backgroundColor:'blue',width:80,height:30,
marginBottom:30}}></View>
                    <View style={{backgroundColor:'black',width:30,height:30,
marginRight:30}}></View>
                </View>
            );
        }
    }
```

运行工程，效果如图 8-12 所示。

与 margin 相关属性对应，你也可以通过 padding 相关属性设置组件的内边距，组件的内边距会影响其内子组件的布局位置，示例代码如下：

```
    export class LayoutDemoThree extends Component{
        render(){
            return(
               <View style={{height:300,backgroundColor:'yellow',paddingTop:30,
paddingLeft:30}}>
                    <View style={{backgroundColor:'red',width:30,
height:30}}></View>
                    <View style={{backgroundColor:'green',width:30,
height:30}}></View>
                    <View style={{backgroundColor:'blue',width:80,
height:30}}></View>
                    <View style={{backgroundColor:'black',width:30,
height:30}}></View>
                </View>
            );
        }
    }
```

运行工程，效果如图 8-13 所示。

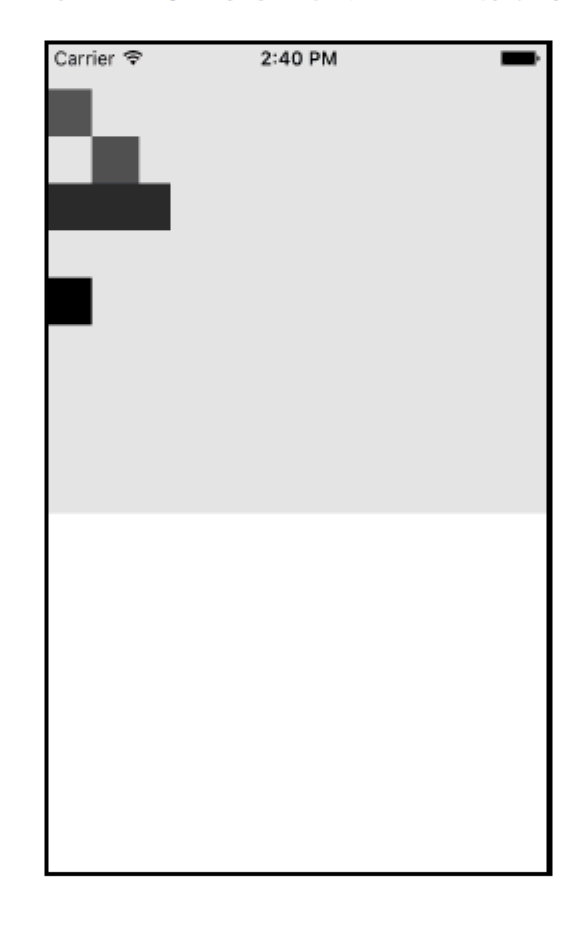


图 8-12　设置组件的外边距

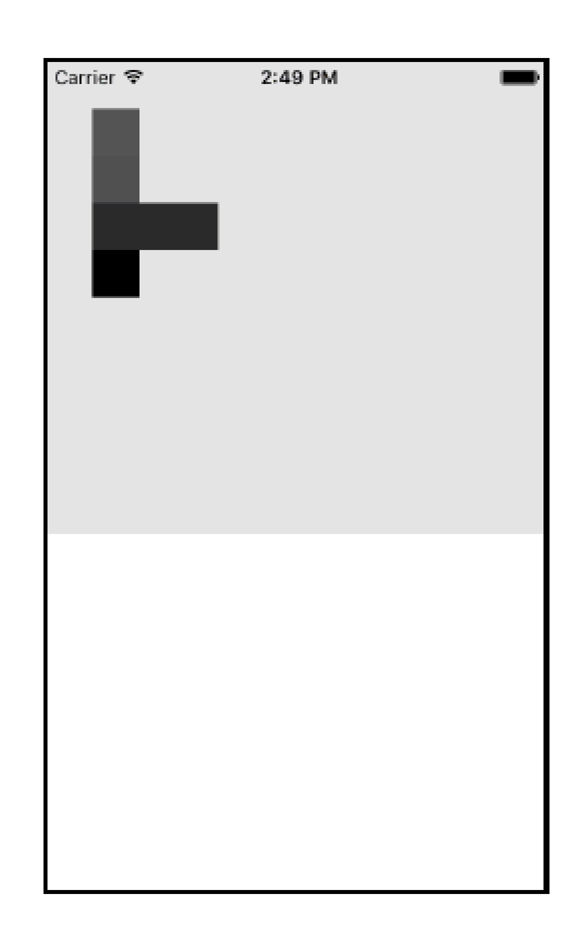


图 8-13　设置组件的内边距

抽 丝 剥 茧

需要注意，如果在定位过程中组件位置有重叠，可以使用 zIndex 属性设置组件的层级，zIndex 的数值越大，视图层级越高，越被优先显示。

8.2 React Native 中的颜色定义

前面已经写了不少 React Native 代码，在定义组件时，通常会设置组件的颜色属性。在 React Native 中，颜色的设置采用的是特殊格式的颜色字符串或十六进制的表示颜色的数值。

在 React Native 中定义颜色有表 8-6 所示的几种格式。

表 8-6 React Native 定义颜色的格式

格式	描述
'#000'	十六进制 RGB 格式，是"#rrggbb"格式的缩写
'#0000'	十六进制 RGBA 格式，是"#rrggbbaa"格式的缩写
'#000000'	十六进制 RGB 格式
'#00000000'	十六进制 RGBA 格式
'rgb(255,255,255)'	十进制 RGB 格式
'rgba(255,255,255,1.0)'	十进制 RGBA 格式
'hsl(360,100%,100%)'	HSL 格式
'hsla(260,100%,100%,1.0)'	HSLA 格式
'transparent'	透明颜色
'red'	颜色名称格式
0x00000000	十六进制数值 RGBA 格式

也可以直接使用颜色名称来定义颜色，React Native 中内置了表 8-7 中列出的可用颜色。

表 8-7 React Native 的可用颜色

名称	颜色	名称	颜色
aliceblue	爱丽丝蓝	lightpink	亮粉
antiquewhite	古董白	lightsalmon	浅肉色
aqua	青	lightseagreen	浅海绿
aquamarine	蓝绿	lightskyblue	浅天蓝
azure	蔚蓝	lightslategray	浅石灰
beige	米黄	lightslategrey	浅石灰
bisque	橘黄	lightsteelblue	浅钢蓝
black	黑	lightyellow	浅黄
blanchedalmond	杏仁白	lime	青柠色

（续表）

名称	颜色	名称	颜色
blue	蓝	limegreen	青柠绿
blueviolet	蓝紫	linen	亚麻色
brown	棕	magenta	品红
burlywood	原木棕	maroon	酱紫
cadetblue	军校蓝	mediumaquamarine	碧绿
chartreuse	黄绿	mediumblue	中蓝
chocolate	巧克力色	mediumorchid	间紫
coral	珊瑚色	mediumpurple	中紫
cornflowerblue	矢车菊蓝	mediumseagreen	中海绿色
cornsilk	米绸色	mediumslateblue	中石板蓝
crimson	深红	mediumspringgreen	中亮绿
cyan	天蓝	mediumturquoise	中宝石绿
darkblue	暗蓝	mediumvioletred	中紫罗兰
darkcyan	深青	midnightblue	深夜蓝
darkgoldenrod	金橘黄	mintcream	奶油色
darkgray	深灰	mistyrose	雾玫瑰色
darkgreen	墨绿	moccasin	卡其
darkgrey	墨灰	navajowhite	军装白
darkkhaki	深卡其	navy	海军蓝
darkmagenta	深洋红	oldlace	浅米色
darkolivegreen	暗橄榄绿	olive	橄榄色
darkorange	深橘	olivedrab	深绿褐色
darkorchid	深紫	orange	橙色
darkred	深红	orangered	橙红色
darksalmon	深橙红	orchid	兰花色
darkseagreen	深海绿	palegoldenrod	浅橘黄色
darkslateblue	深板岩蓝	palegreen	仓绿色
darkslategray	深石板灰	paleturquoise	翠绿色
darkslategrey	暗石板灰	palevioletred	青紫罗兰色
darkturquoise	暗宝石绿	papayawhip	番木色
darkviolet	暗紫罗兰	peachpuff	桃色
deeppink	深粉色	peru	秘鲁褐
deepskyblue	深天蓝色	pink	粉色
dimgray	暗灰	plum	紫红色
dimgrey	暗灰	powderblue	粉蓝色
dodgerblue	闪蓝色	purple	紫
firebrick	火砖红	rebeccapurple	深紫
floralwhite	花白	red	红

（续表）

名称	颜色	名称	颜色
forestgreen	森林绿	rosybrown	褐玫瑰红
fuchsia	紫红	royalblue	宝蓝
gainsboro	淡灰	saddlebrown	重褐色
ghostwhite	幽灵白	salmon	竹红
gold	金色	sandybrown	黄褐
goldenrod	浓黄	seagreen	海绿色
gray	灰	seashell	贝壳色
green	绿	sienna	黄土色
greenyellow	绿黄	silver	银
grey	灰	skyblue	天蓝
honeydew	蜜瓜色	slateblue	石蓝
hotpink	深粉	slategray	石灰
indianred	印度红	slategrey	石灰
indigo	靛蓝	snow	雪白
ivory	象牙色	springgreen	青绿
khaki	土黄	steelblue	钢青色
lavender	淡紫	tan	棕黄
lavenderblush	淡紫红	teal	蓝绿
lawngreen	草坪绿	thistle	蓟色
lemonchiffon	柠檬绸	tomato	番茄色
lightblue	亮蓝	turquoise	松绿色
lightcoral	亮珊瑚	violet	蓝紫
lightcyan	亮天蓝	wheat	麦黄
lightgoldenrodyellow	亮金黄	white	白
lightgray	亮灰	whitesmoke	烟白
lightgrey	亮灰	yellow	黄
lightgreen	亮绿	yellowgreen	黄绿

8.3 警告弹窗的应用

当用户在进行敏感操作时，开发者往往需要通过弹出警告框的方式来让用户对自己的操作进行确认，在 React Native 中提供了两种组件来弹出警告框：跨平台的 Alert 组件和 iOS 平台专用的 AlertIOS 组件。

8.3.1 Alert 组件的应用

Alert 是一个跨平台的警告框，在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 AlertDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {View,Button,Alert} from 'react-native';
export default class AlertDemo extends Component{
    render(){
        return(
            <View>
                <Button height={30} title="ALERT"
                onPress={()=>{
                    Alert.alert(
                        "警告",
                        "天干物燥，小心火烛",
                        [
                            {text: '进入', onPress: () => console.log('come
on')},
                        {text: '取消', onPress: () => console.log('cancel'),
style: 'cancel'},
                        {text: 'OK', onPress: () => console.log('OK Pressed')},
                        ]
                    );
                }}/>
            </View>
        );
    }
}
```

上面的代码创建了一个测试按钮，当单击按钮时，进行警告框的弹出，运行工程，在 iOS 平台与 Android 平台的效果分别如图 8-14 与图 8-15 所示。



图 8-14 iOS 平台的警告框



图 8-15 Android 平台的警告框

在弹出警告框时，只需要直接调用 Alert.alert()函数即可，这个函数中最多可以接收 4 个参数，解释如表 8-8 所示。

表8-8 Alert.alert()函数参数

参数	意义
第 1 个参数	设置警告框的标题，设置为字符串
第 2 个参数	设置警告框的内容文案，设置为字符串

第 3 个参数	设置警告框中提供的功能按钮，需要设置为按钮数组，其中提供的按钮对象可以设置的属性如下：
	{
	text：按钮标题
	onPress：按钮触发方法
	style：按钮风格（仅 iOS 有效）
	}

第 4 个参数	配置参数（仅 Android 有效）

需要注意，对于 iOS 平台，功能按钮的数量可以设置任意多个，并且可以为它们提供一个 style 属性来设置风格。style 属性可选值有 cancel 和 destructive，分别表示取消风格的按钮和消极风格的按钮。在 Android 平台，你最多可以提供 3 个功能按钮，并且在 Android 设备上，当警告框弹出时，如果用户单击屏幕，警告框也会默认消失，可以通过提供第 4 个配置参数来控制在警告框弹出用户单击屏幕的行为，示例代码如下：

```
    <Button height={30} title="ALERT"
     onPress={()=>{
        Alert.alert(
            "警告",
            "天干物燥，小心火烛",
            [
                {text: '进入', onPress: () => console.log('come on')},
                {text: '取消', onPress: () => console.log('cancel'), style:
'cancel'},
                {text: 'OK', onPress: () => console.log('OK Pressed')},
            ],
            {
                cancelable:true,
                onDismiss:()=>{
                    console.log("dismiss");
            }
        }
     );
    }}/>
```

其中 cancelable 属性设置为 true 表示允许用户单击屏幕取消警告框，设置为 false 表示不允许用户单击屏幕取消警告框，onDismiss 属性用来设置用户单击屏幕取消警告框行为的回调函数。

8.3.2　iOS 平台专用警告框 AlertIOS

AlertIOS 是 iOS 平台专用的警告框，不仅可以像 Alert 一样展示普通的功能按钮，还可以添加文本输入框。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 AlertIOSDemo.js 的文件，在其中编写如下代码：

```
    import React,{Component} from 'react';
    import {View,Button,AlertIOS} from 'react-native';
    export default class AlertDemo extends Component{
        render(){
            return(
                <View>
                    <Button height={30} title="普通的警告框"
                    onPress={()=>{
                        AlertIOS.alert(
                            "警告",
                            "天干物燥，小心火烛",
                            [
                                {text: '进入', onPress: () => console.log('come
on')},
                            {text: '取消', onPress: () => console.log('cancel'),
style: 'cancel'},
                            {text: 'OK', onPress: () => console.log('OK Pressed')},
                            ]
                        );
                    }}/>
                    <Button height={30} title="带文本框的警告框"
                    onPress={()=>{
                        AlertIOS.prompt(
                            "警告",
                            "天干物燥，小心火烛",
                            (textObj)=>{
                                console.log(textObj);
                            },
                            'login-password',
                            'www.abcd@163.com',
                            'email-address'
                        );
                    }}/>
                </View>
            );
        }
    }
```

上面的代码创建了两个按钮，第 1 个按钮弹出普通的警告框，其用法基本和 Alert 组件一致，

第 2 个按钮弹出带文本框的警告框，其参数意义如表 8-9 所示。

表8-9　警告框参数

参数	意义
第 1 个参数	警告框标题，需要设置为字符串
第 2 个参数	警告框内容，需要设置为字符串
第 3 个参数	单击警告框 OK 按钮后回调的函数，其中会将文本框中的内容传入
第 4 个参数	设置警告框的类型，可选枚举字符串如下： • default：默认模式，无文本框 • plain-text：普通文本框模式 • secure-text：密码框模式 • login-password：登录账户密码框模式
第 5 个参数	设置第一个文本框中的默认文字
第 6 个参数	设置弹出键盘类型，可选枚举如下： • default • email-address • numeric • phone-pad • ascii-capable • numbers-and-punctuation • url • number-pad • name-phone-pad • decimal-pad • twitter • web-search

运行工程，带文本框的警告框效果如图 8-16 所示。



图 8-16　带文本框的警告框

8.4　ActionSheetIOS 抽屉视图的应用

抽屉视图是 React Native 在 iOS 平台特有的一种视图组件，表现为从界面下方上滑出抽屉卡。ActionSheetIOS 支持两种模式的抽屉视图，即普通功能列表抽屉与分享视图抽屉。普通功能列表的抽屉展示一组功能按钮，用户单击某个按钮后开发者可以实现自己的业务逻辑，分享视图抽屉的功能则更加定向，专门用来进行“社交分享功能”。

8.4.1　普通功能列表抽屉

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ActionSheetIOSDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {ActionSheetIOS,View,Button} from 'react-native';
export default class ActionSheetIOSDemo extends Component{
    render(){
        return(
            <View>
                <Button title="ActionSheet Normal" onPress={()=>{
                    ActionSheetIOS.showActionSheetWithOptions({
                        options:["title1","title2","title3","delete",
"cancel"],
                        cancelButtonIndex:4,
                        destructiveButtonIndex:3,
                        title:"ActionSheetIOS",
                        message:"message info"
                        },(index)=>{
                            console.log(index);
                        });
                }}/>
            </View>
        );
    }
}
```

上面代码中创建了一个按钮，单击按钮后会弹出一组功能列表，效果如图 8-17 所示。showActionSheetWithOptions 方法中需要传入两个参数，第一个参数为配置对象，第二个参数为用户单击抽屉中某个选项后回调的函数，其中会将选项按钮所对应的下标传入。配置对象中可用属性如表 8-10 所示。

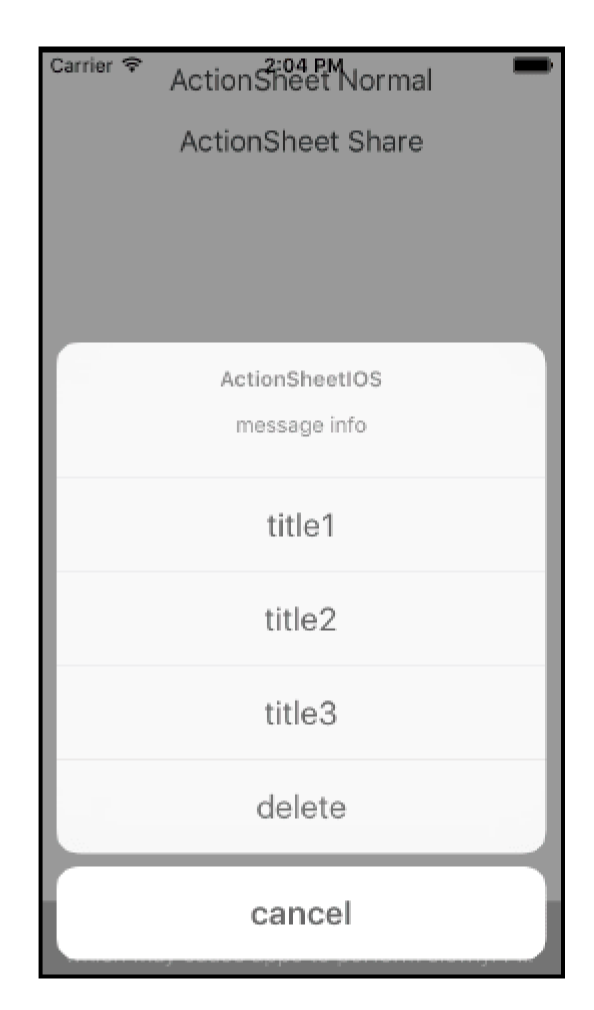


图 8-17 功能列表风格的 ActionSheetIOS

表 8-10 showActionSheetWithOptions 方法参数配置对象的属性

属性	描述
options	设置抽屉列表功能按钮标题，需要设置为字符串数组
cancelButtonIndex	设置取消状态按钮所在的下标
destructiveButtonIndex	设置消极状态按钮所在的下标
title	设置抽屉标题
message	设置抽屉内容文案

8.4.2 分享视图抽屉

许多移动应用都会接入一些社会化分享的功能，例如可以将你感兴趣的网页、喜欢的图片通过社交平台、邮件或短信分享给朋友。iOS 系统自带一套完整的分享组件，可以使用 showShareActionSheetWithOptions 方法来弹出分享视图，示例代码如下：

```
import React,{Component} from 'react';
import {ActionSheetIOS,View,Button} from 'react-native';
export default class ActionSheetIOSDemo extends Component{
    render(){
        return(
            <View>
                <Button title="ActionSheet Share" onPress={()=>{
                    ActionSheetIOS.showShareActionSheetWithOptions({
                        message:"share info",
                        url:"http://www.baidu.com",
                        subject:"image Share",
                        excludedActivityTypes: [
                    'com.apple.UIKit.activity.PostToTwitter'
                        ]
```

```
                    },(fail)=>{
                        console.log("fali"+fail);
                    },(success)=>{
                        console.log("success"+success);
                    });
                }}/>
            </View>
        );
    }
}
```

showShareaActionSheetWithOptions 方法需要传入 3 个参数，如表 8-11 所示。

表 8-11　showShareaActionSheetWithOptions 参数

参数	描述

第 1 个参数	配置对象，可设置的属性如下：
	{
	message: 分享的内容
	url: 分享的网址或本地素材地址
	subject: 分享的主题
	excludedActivityTypes: 不包含的分享类型
	}

第 2 个参数	设置调用分享视图出错的回调
第 3 个参数	设置调用分享视图成功的回调，其中会将是否分享成功作为参数传递

关于 excludedActivityType 属性，其需要设置为字符串数组，数组中提供不包含的分享类型，如果想对 iOS 默认提供的分享平台有更多了解，可以在互联网上学习 iOS 原生的 UIActivityType 类型。分享视图效果如图 8-18 所示。

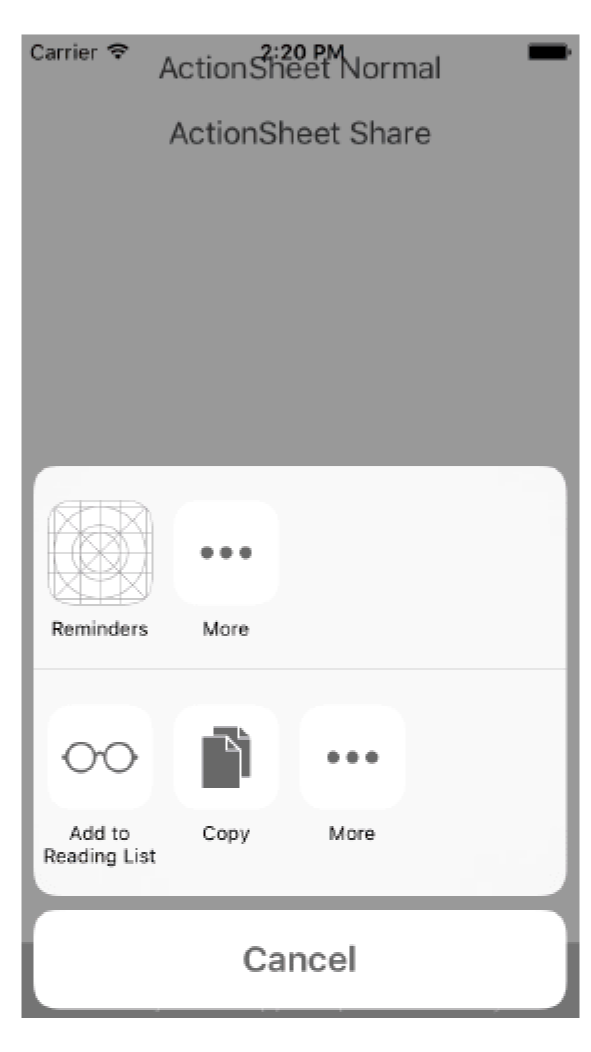


图 8-18　分享视图抽屉

8.5 自定义组件的属性与使用样式表

属性对于组件来说非常重要，如果没有属性，就无法对组件进行个性化的配置，也无法让组件对用户的交互进行反应。前面自定义了不少组件，自定义组件的核心就是将 React Native 提供的基础组件进行组合或嵌套，实现出表现更加丰满、功能更加复杂的组件。对于组件风格样式的设置，前面我们直接通过定义风格样式对象来完成，其实 React Native 中的 StyleSheet 就是专门用来定义样式表的。

8.5.1 自定义组件的属性

在面向对象编程思想中，封装是一种十分重要的特性。在进行自定义 React Native 组件时，将一些设置接口暴露在外，通过设置属性来对自定义组件进行设置是一种十分常用的组件设计模式。在组件中添加的自定义属性会自动被 React Native 整合进组件的 props 对象中去。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 PropsDemo.js 的文件，在其中编写如下代码：

```
    import React,{Component} from 'react';
    import {View,Text} from 'react-native';
    export default class PropsDemo extends Component{
        render(){
            return(
                <View backgroundColor={this.props.bgColor}>
                    <Text style={{marginTop:100,textAlign:'center',
fontSize:24}}>{this.props.title}</Text>
                </View>
            );
        }
    }
```

上面代码中外层容器视图的背景色与文本内容都是通过组件自身属性来进行设置的，将 index.ios.js 与 index.android.js 文件修改如下：

```
    import PropsDemo from './Demo/PropsDemo';
    export default class HelloWorld extends Component {
      render() {
        return (<PropsDemo bgColor="red" title="我是标题"/>);
      }
    }
    AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

运行工程，效果如图 8-19 所示。

通过对组件的属性进行自定义，大大增加了组件的灵活性与可复用性。

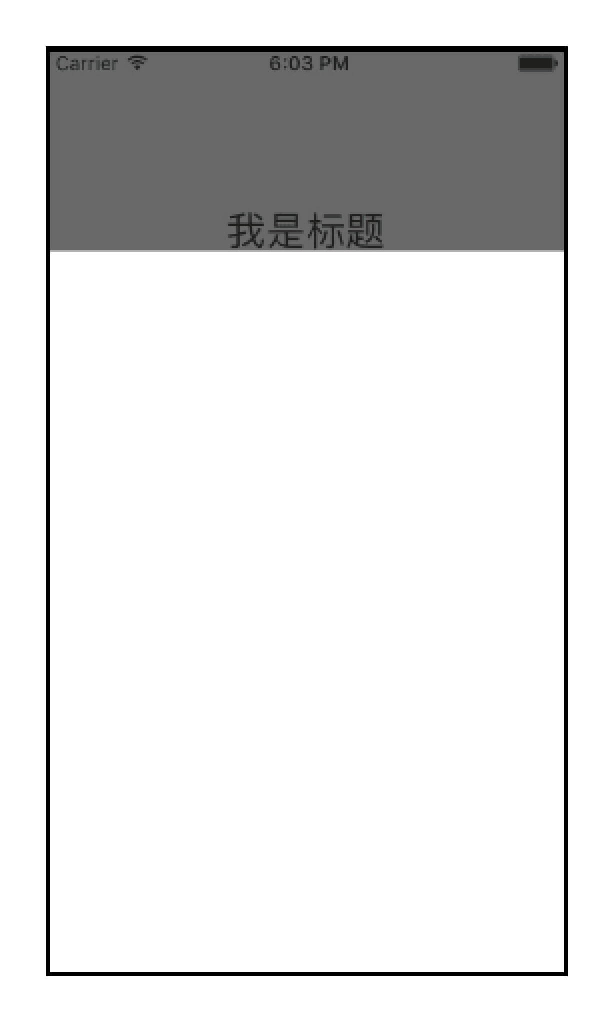


图 8-19　自定义组件属性

8.5.2　通过 StyleSheet 样式表定义组件的风格

通过 StyleSheet 可以定义一组风格样式对象，使用这种方式定义的风格对象可以方便地进行管理与复用。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 StyleSheetDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {View,StyleSheet} from 'react-native';
export default class StyleSheetDemo extends Component{
    render(){
        return(
            <View style={style.container}>
                <View style={style.view1}></View>
                <View style={style.view2}></View>
                <View style={style.view3}></View>
            </View>
        );
    }
}
//定义样式表
let style = StyleSheet.create({
    container:{
        flex:1,
        backgroundColor:'red'
    },
    view1:{
        height:30,
        marginTop:10,
        backgroundColor:'blue'
    },
```

```
    view2:{
        height:50,
        marginTop:20,
        backgroundColor:'green'
    },
    view3:{
        height:40,
        marginTop:20,
        backgroundColor:'yellow'
    }
});
```

修改 index.ios.js 与 index.android.js 文件后运行工程，效果如图 8-20 所示。

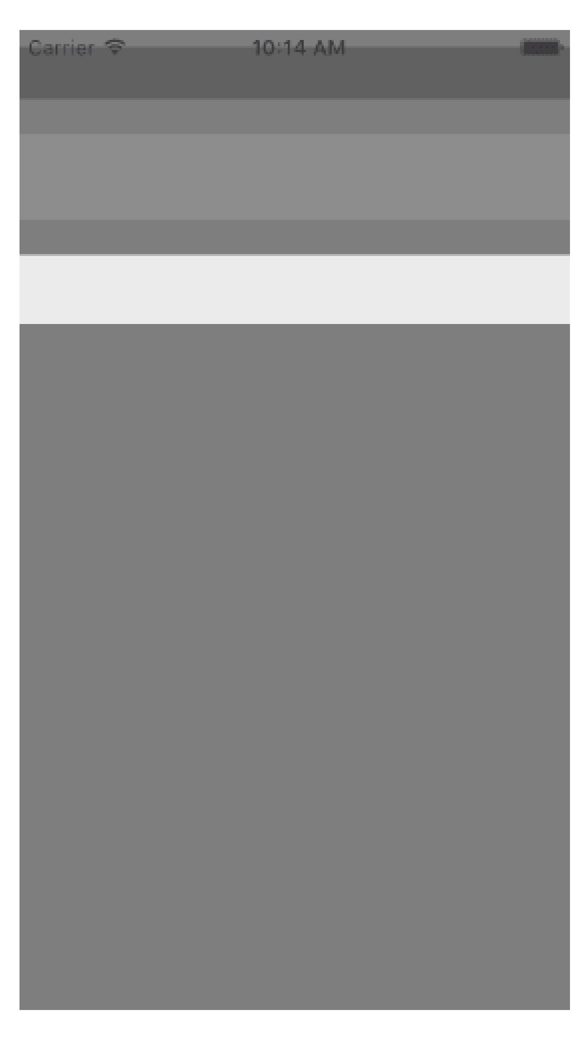


图 8-20　使用 StyleSheet 定义样式表

抽 丝 剥 茧

在实际开发中，建议组件的样式定义都采用 StyleSheet 样式表这种方式：首先，将对象定义抽离到组件标签之外，可以使代码更加清晰、结构更加简洁；其次，这种方式可以对样式对象进行复用，减少性能的开销；最后，React Native 官方团队也声明之后会继续优化样式表的渲染速度，因此应该尽量使用这种方式。

8.6　Android 平台的时间选择器

针对 Android 平台，React Native 提供了 TimePickerAndroid 类，用来弹出一个时间选择器对话框。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 TimePickerAndroidDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {TimePickerAndroid,View,Button} from 'react-native';
export default class TimePickerAndroidDemo extends Component{
    render(){
        return(
            <View>
                <Button title="Time" onPress={()=>{
                    TimePickerAndroid.open(
                        {
                            hour:22,
                            minute:30,
                            is24Hour:true
                        }).then(({minute,hour,action})=>{
                            console.log("success",minute,hour,action);
                        },(fail)=>{
                            console.log("fail",fail);
                        });
                }}/>
            </View>
        );
    }
}
```

TimePickerAndroid 类的静态方法 open 中需要传入一个配置对象，用来对时间选择器进行初始化，其中可以定义的属性如表 8-12 所示。

需要注意，open 方法会返回一个 Promiss 对象，上面代码中 Promiss 的 then 链中的第 1 个参数设置当用户选择时间完成后的回调（第 2 个参数设置了时间选择器弹出失败的回调），回调中会传入一个包含 minute、hour 和 action 属性的参数，属性意义如表 8-13 所示。

表 8-12　open 方法传入配置对象的属性

属性	说明
hour	设置小时（0～23）
minute	设置分钟（0～59）
is24Hour	设置是否为 24 小时制（布尔值）

表 8-13　回调参数

参数	说明
minute	用户选择的分钟，如果用户取消选择，这个属性为 undefined
hour	用户选择的小时，如果用户取消选择，这个属性为 undefined
action	用户的行为。如果选择了时间，这个值为 TimePickerAndroid.timeSetAction；如果用户取消了选择，这个值为 TimePickerAndroid. dismissedAction

修改 index.android.js 文件后，运行工程，效果如图 8-21 所示。



图 8-21　Android 平台的时间选择器

抽 丝 剥 茧

关于 Promiss 承诺对象构建任务链的内容，在 ECAMScript 基础章节有详细的介绍，如果你对本节的内容感觉有些陌生，建议你回到前面的语法部分温习一下。本节是 Promiss 对象实战应用的最好示例。

8.7　Android 平台悬浮提示信息 Toast 的应用

原生的 Android 开发经常会使用到 Toast 来弹出简单的悬浮提示信息。React Native 中的 ToastAndroid 类用来在 Android 平台弹出悬浮提示信息。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ToastAndroidDemo.js 的文件，在其中编写如下代码：

```
    import React,{Component} from 'react';
    import {View,Button,ToastAndroid} from 'react-native';
    export default class ToastAndroidDemo extends Component{
        render(){
            return(
                <View>
                    <Button title="Toast Normal"
                            onPress={()=>{
                                ToastAndroid.show("message info",
ToastAndroid.SHORT);
                            }}/>
                    <Button title="Toast Gravity"
```

```
                            onPress={()=>{
                                ToastAndroid.showWithGravity("message info",
ToastAndroid.LONG,ToastAndroid.CENTER);
                            }}/>
                </View>
            );
        }
    }
```

ToastAndroid 类中提供了两个方法来弹出悬浮提示，show 方法可以接收两个参数，第 1 个参数为要显示的提示信息，第 2 个参数为提示显示的时长设置，可选值有 ToastAndroid.SHORT 与 ToastAndroid.LONG。showWithGravity 方法也是用来弹出悬浮提示的，不同的是 show 方法弹出的提示默认显示在屏幕下方，showWithGravity 方法多出一个参数设置提示的显示位置，可设置的值有 ToastAndroid.CENTER、ToastAndroid.TOP 和 ToastAndroid.BOTTOM。

运行工程，效果如图 8-22 所示。

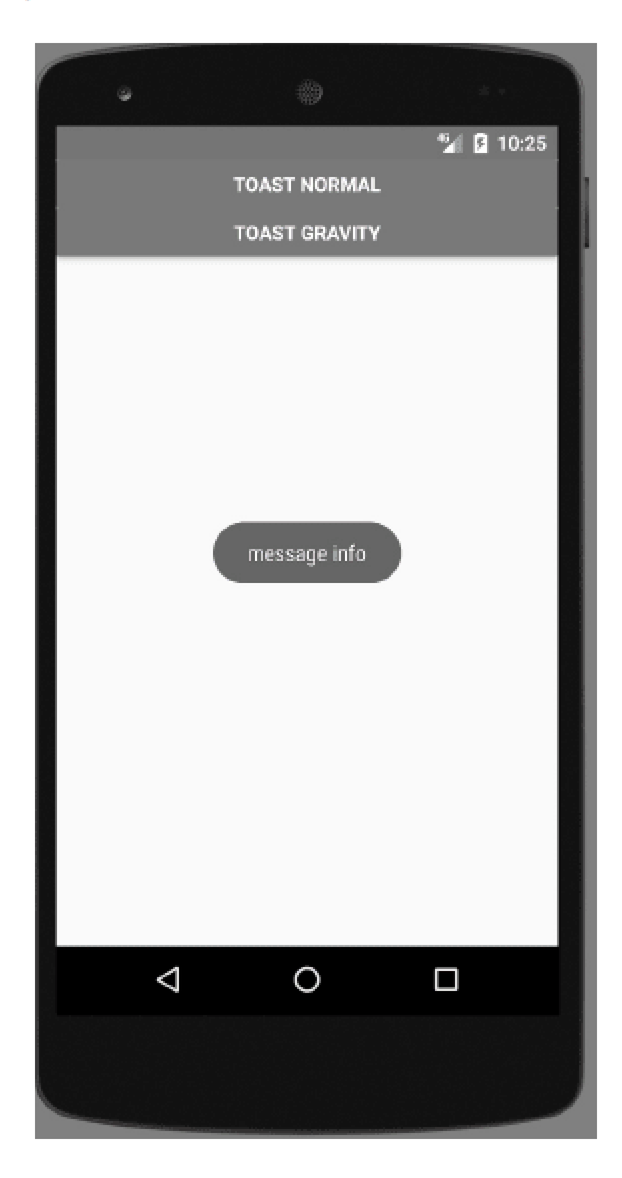


图 8-22　ToastAndorid 悬浮提示

8.8　监听与控制 Android 设备返回键的行为

和 iOS 设备不同的是，几乎所有的 Android 设备都会带一个返回键，通常情况下用户单击返回按钮可以实现界面的返回或者退出应用程序。在 React Native 开发中，你也可以监听用户单击返回键的动作来实现自己的逻辑。

监听用户单击返回键的功能由 React Native 中的 BackAndroid 类提供，这个类中提供了 3 个静态方法，如表 8-14 所示。

表 8-14 BackAndroid 类的静态方法

方法名	解释	平台	参数
addEventListener	这个方法用来添加用户单击返回按钮的监听事件	Android	需要传入两个参数，第 1 个参数为固定字符串"hardwareBackPress"，表示监听设备返回键的按下行为（目前只支持监听这一种行为），第 2 个参数为回调函数，这个函数需要返回一个布尔值，如果返回 true 表示中断监听链，返回 false 表示不中断监听链
removeEventListener	这个方法用来移除一个监听事件	Android	参数意义和 addEventListener 方法完全一致
exitApp	调用这个方法直接退出应用程序	Android	无

上面有提到监听链的概念，实际上你可以多次调用 BackAndroid 类的 addEventListener 方法来添加多个监听回调，当用户单击 Android 设备的返回按钮后，系统会一次按照监听方法的添加顺序逆序调用监听函数，其中如果有返回 true，则中断此调用链。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 BackAndroidDemo.js 的文件，在其中编写如下代码：

```
    import React,{Component} from 'react';
    import {BackAndroid,View,Button} from 'react-native';
    export default class BackAndroidDemo extends Component{
        constructor(props){
            super(props);
            BackAndroid.addEventListener("hardwareBackPress",this.hander);
        }
        render(){
            return(
                <View>
                    <Button title="remove" onPress={()=>{
                        BackAndroid.removeEventListener("hardwareBackPress",
this.hander);
                    }}/>
                    <Button title="EXIT" onPress={()=>{
                        BackAndroid.exitApp();
                    }}/>
                </View>
            );
        }
        hander(){
            console.log("goBack");
            return true;
        }
    }
```

运行工程，打开调试模式，根据打印信息可以看到对用户单击设备返回按钮的监听效果。

8.9　监听程序运行状态

一款应用程序在运行时总会有许多种状态，例如在用户使用时应用程序通常处于前台运行状态，当用户回到桌面或者启动其他应用程序时，这款应用程序就处于后台挂起状态（当然，在前台运行切换到后台挂起之间还会有一个瞬时的中间状态）。当设备内存不足时，应用程序也会发出内存不足的状态提醒。

在 React Native 中，你可以使用 AppState 类来监听应用程序的前后台状态切换，也可以监听内存不足时系统发出的警告。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 AppStateDemo.js 的文件，在其中编写如下测试代码：

```
import React,{Component} from 'react';
import {AppState,View,Button} from 'react-native';
export default class AppStateDemo extends Component{
    constructor(props){
        super(props);
        AppState.addEventListener("change",this.handler1);
    }
    render(){
        return(
            <View>
                <Button title="addHandler2" onPress={()=>{
                    AppState.addEventListener("change",this.handler2);
                }}/>
                <Button title="removeHandler2" onPress={()=>{
                    AppState.removeEventListener("change",this.handler2);
                }}/>
            </View>
        );
    }
    handler1(){
        console.log("handler1",AppState.currentState);
        return false;
    }
    handler2(){
        console.log("handler2",AppState.currentState);
        return true;
    }
}
```

AppState 类的 addEventListener 方法用来添加一个程序状态的监听，这个方法中第 1 个参数设置监听的内容，可设置的字符串为“change”或“memoryWarning”，设置为“change”表示监听程序的前后台状态，设置为“memoryWarning”表示监听内存警告通知。AppState 类的静态属性 currentState 用来获取当前程序的前后台状态，其中可能值列举如表 8-15 所示。

表 8-15 currentState 属性值

属性值	说明
active	应用正在前台运行
background	应用正在后台运行
inactive	前台切换到后台时的过渡状态

需要注意，你可以添加多个监听回调，当应用程序状态改变时，系统会按照添加顺序依次执行回调函数。同样，使用 removeEventListener 方法可以移除一个已经添加的监听回调。

8.10 跨平台的分享功能

关于社会化分享，在学习 ActionSheetIOS 时你已经有所了解，ActionSheetIOS 可以创建分享模式的活动列表，但是只能应用于 iOS 平台，本节将介绍在 React Native 中专门用于分享功能的跨平台组件 Share。

在需要用户进行分享操作时，你可以直接调用 Share 类的 share 静态方法，这个方法在 iOS 平台会弹出一个分享活动列表，在 Android 平台会弹出一个分享对话框，这个方法中需要传入两个参数，都是对象类型，对象可配置的属性如表 8-16 所示。

表8-16 share静态方法传入参数

参数	属性

第 1 个参数	{
	message：分享的内容
	title：分享的标题
	url：分享的链接，仅 iOS 平台可用
	}

第 2 个参数	{
	excludedActivityTypes：数组，设置不包含的默认分享平台，仅 iOS 平台有效，可参见 ActionSheetIOS 一节
	tintColor：仅 iOS 平台有效
	dialogTitle：对话框标题，仅 Android 平台有效
	}

调用 share 方法后会返回一个 Promiss 承诺对象，对于 iOS 平台，Promiss 对象的 then 链中会传入一个包含 action 属性和 actionType 属性的对象，通知开发者用户分享的行为。action 属性对应的值定义在 Share 类中，可以使用表 8-17 中的 Get 方法来获取。

表8-17 获取action属性值的Get方法

Get 方法	说明
shareAction()	用户进行了分享行为
dismissedAction()	用户取消了分享行为

需要注意，在 Android 平台上，无论用户是进行了分享还是取消了分享，Pormiss 中都会传入 shareAction 行为，也就是说，在 Android 平台上，Share 组件并不能区分出用户是否确实进行了分享。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ShareDemo.js 的文件，在其中编写如下代码：

```
import React,{Component} from 'react';
import {View,Share,Button} from 'react-native';
export default class ShareDemo extends Component{
    render(){
        return(
        <View>
            <Button title="share" onPress={()=>{
                Share.share({
                    message:"分享的内容",
                    title:"title",
                    url:"https://www.baidu.com"
                },{
                    tintColor:'red',
                    dialogTitle:'dialogTitle'
                }).then((action)=>{
                    console.log("success",action);
                },(action)=>{
                    console.log("error",action);
                });
            }}/>
        </View>
        );
    }
}
```

修改 index.ios.js 文件与 index.android.js 文件后，运行工程，在两个平台上的效果分别如图 8-23 与图 8-24 所示。

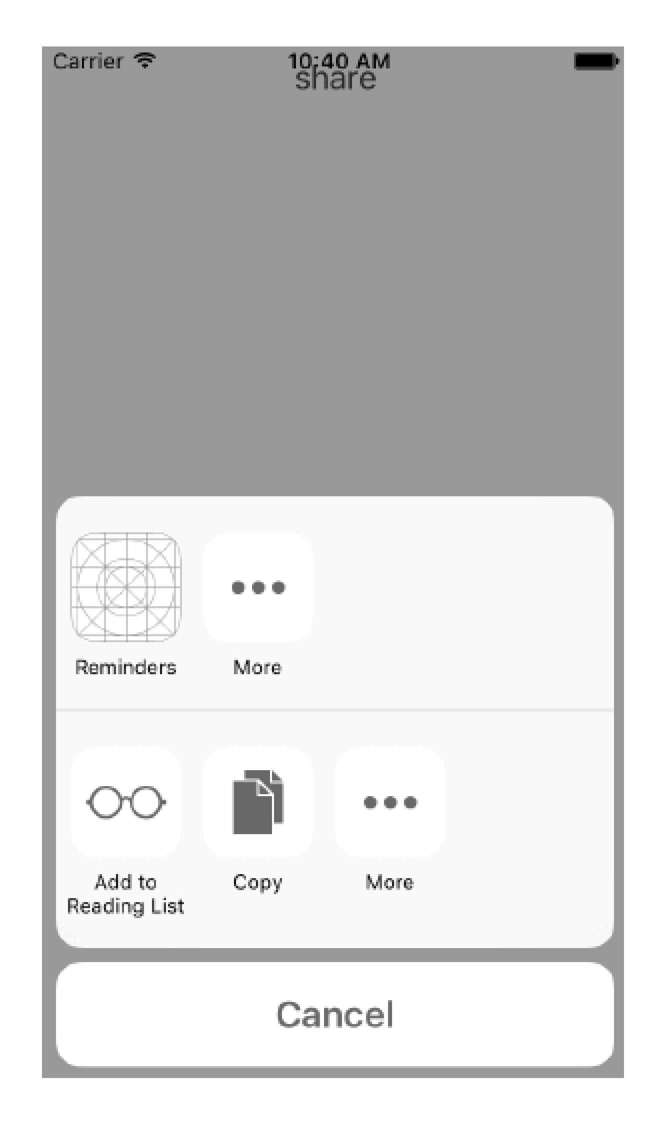



图 8-23　iOS 平台的分享功能

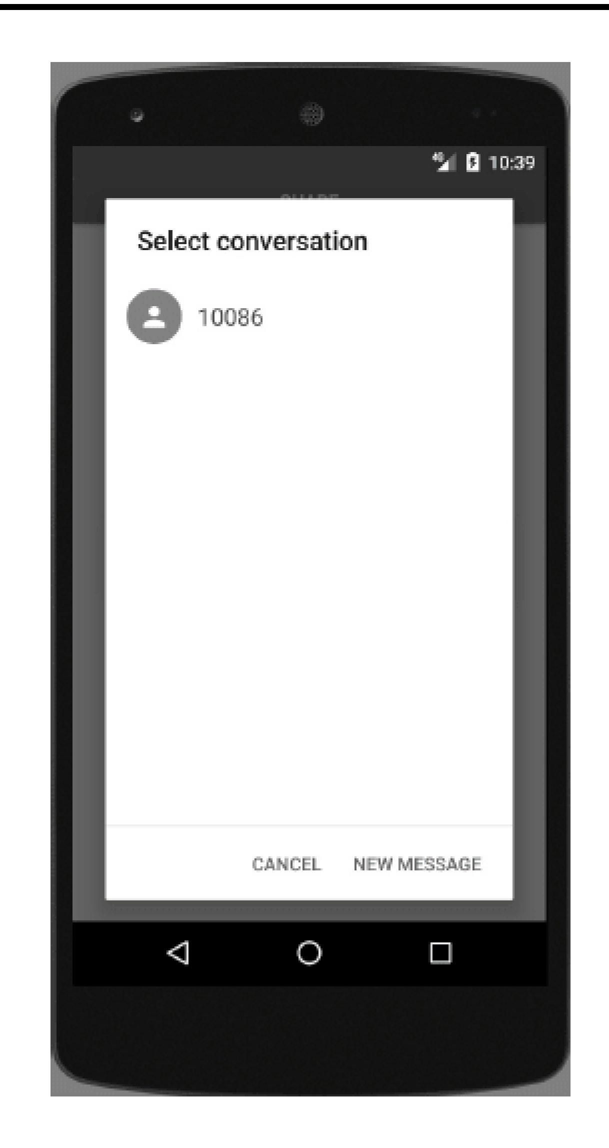


图 8-24　Android 平台的分享功能

8.11　监听键盘事件

React Native 中的许多组件都有弹出虚拟键盘的功能，例如 TextInput。在实际开发中，很多时候你需要监听键盘的行为来做业务逻辑。React Native 中提供了 Keyboard 对象用来添加键盘行为的监听。

Keyboard 对象中定义了 4 个方法，如表 8-18 所示。

表 8-18　Keyboard 对象中的 4 个方法

方法名	解释	平台	参数
addListener	添加一个键盘行为的监听	通用	需要传入两个参数：(nativeEvent,func) • nativeEvent：字符串，要监听的键盘行为 • func：函数回调方法
removeAllListeners	移除某个键盘行为的所有监听方法	通用	需要传入 nativeEvent 参数
removeListener	移除某个监听回调	通用	需要传入两个参数：(nativeEvent,func) • nativeEvent：字符串，键盘行为 • func：移除的函数回调方法
dismiss	直接收起键盘	通用	无

表 8-18 中的 nativeEvent 是 React Native 中定义的一组字符串用来表示键盘的行为，如表 8-19 所示。

表 8-19　React Native 中定义的字符串对应的键盘行为

名称	解释	平台
keyboardWillShow	键盘将要展现	iOS
keyboardDidShow	键盘已经展现	通用
keyboardWillHide	键盘将要隐藏	iOS
keyboardDidHide	键盘已经隐藏	通用
keyboardWillChangeFrame	键盘将要改变位置	iOS
keyboardDidChangeFrame	键盘已经改变位置	iOS

关于两个键盘改变位置的行为，回调函数中会将键盘的起始与结束位置封装为对象传入。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 KeyboardDemo.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import { Keyboard, TextInput,Button,View } from 'react-native';
export default class KeyboardDemo extends Component{
    constructor(props){
        super(props);
        Keyboard.addListener('keyboardWillShow',()=>{
            console.log('keyboardWillShow1');
        });
        Keyboard.addListener('keyboardWillShow',()=>{
            console.log('keyboardWillShow2');
        });
        Keyboard.addListener('keyboardDidShow',()=>{
            console.log('keyboardDidShow');
        });
        Keyboard.addListener('keyboardWillHide',()=>{
            console.log('keyboardWillHide');
        });
        Keyboard.addListener('keyboardDidHide',()=>{
            console.log('keyboardDidHide');
        });
        Keyboard.addListener('keyboardWillChangeFrame',(keyboard)=>{
            console.log('keyboardWillChangeFrame',keyboard);
        });
        Keyboard.addListener('keyboardDidChangeFrame',()=>{
            console.log('keyboardDidChangeFrame');
        });
    }
    render(){
        return(
```

```
                <TextInput style={{height:30,borderColor:'red',
borderWidth:1,top:30}} onSubmitEditing={()=>{
                    Keyboard.dismiss();
                }}/>
            );
        }
    }
```

修改 index.ios.js 与 index.android.js 文件后，运行调试模式即可看到打印效果。需要注意，对于同一种键盘行为，你可以添加多个监听回调，系统会按照添加顺序依次执行回调方法。

8.12 React Native 网络技术

互联网技术越来越发达，移动应用越来越离不开网络。网络技术已经成为应用程序至关重要的一部分，资讯类应用需要通过网络来获取资讯信息、社交类应用需要通过网络来传递消息、电商类应用则需要通过网络实现购物结算，就连单机游戏有时也需要通过网络来进行得分分享、道具购买等。React Native 中提供了访问网络的方法和接口，其中最重要的访问网络方式有两种：fetch 请求与 XMLHttpRequest(AJAX)技术。

8.12.1 使用 fetch 方法进行网络请求

fetch 函数是 React Native 中提供的访问网络的方法，其用法和 Web API 中的 fetch 函数一致。其使用方法十分简单，在调用 fetch 函数时，你需要传递两个参数，其中第 1 个参数为请求地址，第 2 个参数为配置对象，对象中常用的可配置的属性意义如表 8-20 所示。

表 8-20 fetch 函数参数的配置对象

属性名	意义	值类型或可选值
method	设置请求方法	字符串，例如 GET、POST
headers	设置请求头信息	自定义对象
mode	设置请求模式	字符串，如 cors、no-cors
cache	请求的缓存模式	字符串，如 default
body	设置请求参数对象，如果为 POST 方式的请求，这个参数必填	自定义对象

fetch 函数会返回一个 Promiss 承诺对象，在完成的回调中会将请求的返回数据传入。在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 NetDemo.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  Button,
  View
} from 'react-native';
```

```
export default class NetDemo extends Component{
    render(){
        return(
            <View>
                <Button title="Fetch" onPress={()=>{

fetch("https://facebook.github.io/react-native/movies.json",{
                            method:"GET",
                            headers:{
                                name:"PK"
                            }
                        }).then((response)=>{
                            console.log(response);
                            return response.json();
                        }).then((json)=>{
                            console.log(json);
                        });
                }} />
            </View>
        );
    }
}
```

修改 index.ios.js 与 index.android.js 文件后，运行工程，打开调试模式可以看到打印出的请求返回数据。上面示例代码中请求数据的地址为 Facbook 提供的示例接口将会返回一个影单列表。

关于请求返回数据，还需要做深入的了解，上面代码中的 response 实际上是 Response 对象，这个对象中有许多属性用来存储数据信息和方法，对数据进行整理，如表 8-21 所示。

表 8-21　常用属性列表

属性名	释义
type	返回数据类型，只读的
url	请求的地址，只读的
status	请求的状态，只读的
ok	请求是否成功，只读的布尔值
headers	返回头信息，只读的对象
bodyUsed	布尔值，标记数据信息是否被读取过

常用方法如表 8-22 所示。

表 8-22　常用方法

方法名	意义	参数
clone	克隆返回数据对象	无
error	返回一个绑定了异常的 Response 对象	无

（续表）

方法名	意义	参数
arrayBuffer	读取返回数据，并将其置为已读，返回以数组为回调参数的 Promisse 对象	无
blob	读取返回数据，并将其置为已读，返回以 Blob 数据为回调参数的 Promiss 对象	无
json	同上，数据会被作为 JSOM 数据解析成对象	无
text	同上，数据会被解析为字符串	无

抽 丝 剥 茧

需要注意，fetch 方法可能会抛出异常，你可以使用 Promiss 对象调用 catch 方法来进行捕获。

8.12.2 使用 XMLHttpRequest 进行网络请求

XMLHttpRequest 是 Web 开发中常用的网络 API，其为网页提供了客户端与服务端的通信功能。XMLHttpRequest 是 AJAX 技术的一种实现（AJAX 即非同步的 JavaScript 及 XML 技术）。

在 8.12.1 小节新建的 NetDemo.js 文件中添加一个新的按钮，实现如下：

```
    <Button title="ajax" onPress={()=>{
        var request = new XMLHttpRequest();
        request.responseType="json";
        request.onreadystatechange = () => {
            if (request.readyState!==4) {
                return;
            }
            if (request.status === 200) {
            console.log('success', request.response);
            } else {
                console.warn('error');
            }

        };
        request.open('GET',
"https://facebook.github.io/react-native/movies.json");
        request.send();
    }}/>
```

上面的示例代码创建了一个 XMLHttpRequest 对象，responseType 属性设置返回数据的类型，这里的示例请求服务端返回了 JSON 数据，所以这里也设置为“json”。onreadystatechange 属性用来设置当请求状态改变时的回调，readyState 属性为请求的状态，当其为 4 时表示请求数据获取完成。

XMLHttpRequest 对象常用属性如表 8-23 所示。

表 8-23　XMLHttpRequest 对象常用属性

属性名	意义	值类型
onreadystatechange	设置当请求状态改变时的回调	函数
readyState	请求状态码	• 0：未打开，open 方法还未调用 • 1：已经打开，未发送，send 方法还未调用 • 2：已经获取响应头信息 • 3：正在下载数据中 • 4：请求完成
response	请求返回的数据	会根据 responseType 指定的数据类型进行解析
responseText	请求返回的数据为文本时，这个属性存放文本	字符串
responseType	设置请求返回数据的类型	• ""：表示字符串（默认） • "arraybuffer"：数组 • "blob"：Blob 数据 • "json"：JSON 数据 • "text"：字符串
status	请求的响应状态码	只读，例如 200 表示成功
statusText	请求的响应状态文本	字符串

XMLHttpRequest 对象常用方法如表 8-24 所示。

表 8-24　XMLHttpRequest 对象常用方法

方法名	意义	参数
abort	中断正在进行的请求	无
getAllResponseHeaders	获取所有响应头数据	无
getResponseHeader	获取指定响应头数据	字符串
open	打开一个请求	需要指定如下参数(method,url,async,user,password)： • method：设置请求方法，如 GET • url：设置请求地址 • async：可选参数，设置是否为异步请求，设置为 true 则为异步请求，设置为 false 则为同步请求，默认为 true • user：可选参数，用户名 • password：可选参数，密码
send	发送请求	无
setRequestHeader	设置请求头	参数(key,value)如下： • key：请求头中的键 • value：对应的值

8.13 进行用户位置获取

React Native 中封装了定位用户位置信息的服务，使用 Geolocation 对象可以方便地监听用户位置。需要注意，iOS 平台需要在工程的 Info.plist 文件中添加 NSLocationWhenInUseUsageDescription 字段来允许定位服务（使用 react-native init 命令创建的工程自动已经添加），Android 平台需要在工程的 AndroidManifest.xml 文件中添加如下标签获取权限：

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 GeolocationDemo.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  Button,
  View
} from 'react-native';
var Geolocation = require('Geolocation');
export default class GeolocationDemo extends Component{
    render(){
        return(
            <View>
                <Button title="GEO" onPress={()=>{
                    Geolocation.getCurrentPosition((obj)=>{
                        console.log(obj);
                    },()=>{
                        console.log("error");
                    },{
                        timeout:10000,
                        maximunAge:100000,
                        enableHighAccuracy:false,
                        distanceFiter:10
                    });
                }}/>
            </View>
        );
    }
}
```

getCurrentPosition 方法用来获取用户位置，其参数意义如表 8-25 所示。

表8-25　getCurrentPosition参数

参数	意义
参数 1	设置定位成功的回调，回调中会传入定位信息对象，其中包含经纬度、速度、海拔等各种信息，可以在调试模式中打印观察
参数 2	定位失败的回调
参数 3	配置参数，对象中可设置的属性如下： • timeout：设置定位超时时间，单位毫秒 • maximumAge：设置定位信息有效期，单位毫秒 • enableHighAccuracy：设置是否开启精准定位 • distanceFiter：设置位置容差，单位米

除了 getCurrentPosition 方法，Geolocation 对象中还提供了一个 watchPosition 方法，这个方法的参数也是 3 个，意义和 getCurrentPosition 方法完全一致，不同的是，watchPosition 方法实现了对用户位置的实时监听，当用户位置发生改变时，这个方法设置的回调函数都会被调用，同时 watchPosition 方法会返回一个数值作为 ID，可以通过调用 clearWatch 方法传入 ID 作为参数来清除监听。你也可以在定位过程中随时调用 stopObserving 方法来停止定位信息的获取。

博 闻 强 识

对于 iOS 模拟器，可以通过 Debug 工具来模拟位置信息，选择 iOS 模拟器工具栏上的 Debug 选项，选择其中的 Location 选项，可以指定其中某个地点作为模拟器的位置，也可以自定义经纬度来模拟位置（设置后需要重启模拟器生效）。

8.14　数据持久化技术

数据持久化技术在一款应用程序的开发中十分重要。前面所学习的章节中，无论是通过代码在内存中创建出的数据还是通过网络从服务端下载的数据，当用户退出应用程序后，这些数据都会被清空。然而有时候，某些数据需要持久化地保存在用户的手机上，例如用户的登录信息数据、网络缓存数据等。在 React Native 中，进行数据持久化十分容易，React Native 中提供了 key-value 模式的异步存储对象 AsyncStorage。

AsyncStorage 对象是 key-value 模式的异步存储工具。所谓 key-value 模式，是指在进行数据存储时采用的是键值对的模式，一个键对应一个具体的数据值，键不可重复；所谓异步，是指在进行数据写入或读取时，程序流程不受影响。AsyncStorage 对象中提供了丰富的方法，如表 8-26 所示。

表 8-26 AsyncStorage 对象的方法

方法名	解释	平台	参数
setItem	进行数据存储	通用	(key,value,callback(error)) • key：键 • value：值 • callback：存储完成的回调，如果存储失败，会将错误信息作为参数传入
getItem	获取某个键对应的值	通用	(key,callback(error,result)) • key：键 • callback：查询结果回调，会将查询到的数据 result 作为参数传入
removeItem	删除某个键对应的值	通用	(key,callback(error)) • key：键 • callback：删除完成的回调
mergeItem	将某个键对应的值进行 json 合并，需要注意，已经存在的值和要合并的值都必须为严格的 json 字符串	通用	(key,value,callback(error)) • key：键 • value：值 • callback：合并完成后的回调
getAllKeys	获取所有已经存储的键	通用	(callback(error,keys)) • callback：回调，会将所有查询到的键作为参数传入
multiSet	同时设置多个键值	通用	(kvs,callback(error)) • kvs：键值对，采用数组格式，例如[[k1,v1],[k2,v2],...] • callback：完成后的回调
multiGet	同时获取多个键对应的值	通用	(keys,callback(error,results)) • keys：要查询的键组成的数组 • callback：结果回调，会将结果作为参数传入
multiRemove	同时删除多个键对应的值	通用	(keys,callback) • keys：要删除的键组成的数组 • callback：完成的回调
clear	清空所有键值	通用	(callback) • callback：操作完成的回调

在 HelloWorld 项目的 Demo 文件夹下新建一个命名为 AsyncStorageDemo.js 的文件，在其中编写如下测试代码：

```
import React, { Component } from 'react';
import {
  Button,
  View,
```

```
      AsyncStorage
    } from 'react-native';
    export default class AsyncStorageDemo extends Component{
        render(){
            return(
                <View>
                    <Button title="Save" onPress={()=>{
                        AsyncStorage.setItem("name","Jaki",(error)=>{
                            console.log(error);
                        });
                    }}/>
                    <Button title="Get" onPress={()=>{
                        AsyncStorage.getItem("name",(error,result)=>{
                            console.log(error,result);
                        });
                    }}/>
                    <Button title="Remove" onPress={()=>{
                        AsyncStorage.removeItem("name",(error)=>{
                            console.log(error);
                        });
                    }}/>
                    <Button title="AddMerge" onPress={()=>{
                        AsyncStorage.setItem("merge","{\"name\":\"jaki\"}",
(error)=>{
                            console.log(error);
                        });
                    }}/>
                    <Button title="Merge" onPress={()=>{
                        AsyncStorage.mergeItem("merge","{\"age\":24}",
(error)=>{
                            console.log(error);
                        });
                    }}/>
                    <Button title="GetMerge" onPress={()=>{
                        AsyncStorage.getItem("merge",(error,result)=>{
                            console.log(error,result);
                        });
                    }}/>
                    <Button title="GetAllKey" onPress={()=>{
                        AsyncStorage.getAllKeys((error,keys)=>{
                            console.log(error,keys);
                        });
                    }}/>
                    <Button title="mutliSet" onPress={()=>{
```

```
                        AsyncStorage.multiSet([["one","1"],["two","2"]],
(error)=>{
                            console.log(error);
                        });
                    }}/>
                    <Button title="multiGet" onPress={()=>{
                        AsyncStorage.multiGet(["one","two"],(error,results)=>{
                            console.log(error,results);
                        });
                    }}/>
                    <Button title="multiRemove" onPress={()=>{
                        AsyncStorage.multiRemove(["one","two"],(error,
results)=>{
                            console.log(error,results);
                        });
                    }}/>
                    <Button title="clear" onPress={()=>{
                        AsyncStorage.clear((error)=>{
                            console.log(error);
                        });
                    }}/>
                </View>
            );
        }
    }
```

上面的代码演示了 AsyncStorage 对象的所有常用方法，修改 index.ios.js 与 index.android.js 后可以在调试模式下观察效果。

8.15 剪贴板工具的应用

剪贴板是移动应用中常用的复制粘贴数据的工具，React Native 提供了 Clipboard 对象，通过代码来控制复制粘贴行为。

抽 丝 剥 茧
大部分文本输入控件都可以自动响应复制粘贴行为，当用户在 TextInput 上长按时会弹出一个菜单，其中包括文本选择、剪切、复制、粘贴等功能。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 ClipboardDemo.js 的文件，在其中编写如下测试代码：

```
import React, { Component } from 'react';
import {
  TextInput,
  View,
  Text,
  Clipboard,
  Button
} from 'react-native';
export default class ClipboardDemo extends Component{
    render(){
        return(
            <View style={{marginTop:20}}>
                <TextInput
style={{borderWidth:1,borderColor:'gray',height:30}}/>
                <Text onPress={()=>{
                    Clipboard.setString("复制文字");
                }}>复制文字</Text>
                <Button title="打印文字" onPress={()=>{
                    Clipboard.getString().then((str)=>{
                            console.log(str);
                        });
                }}/>
            </View>
        );
    }
}
```

Clipboard 中只提供了两个方法，setString 方法用来将字符串写入剪贴板中，getString 方法用来将剪贴板中的数据读取出来。注意，getString 方法会返回一个 Promise 对象。

8.16　获取设备网络状态

在实际开发中，我们常常需要使用到网络技术，有时还需要获取用户所处的网络环境，例如，在非 WiFi 环境下，为了节省用户流量，一些视频数据往往不会自动下载播放。在 React Native 中，NetInfo 对象提供了异步获取用户网络环境的方法。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 NetInfoDemo.js 的文件，在其中编写如下测试代码：

```
import React, { Component } from 'react';
import {
  Button,
  View,
```

```
    NetInfo
} from 'react-native';
export default class NetInfoDemo extends Component{
    render(){
        return(
            <View>
                <Button title="NetInfo" onPress={()=>{
                    NetInfo.fetch().then((info)=>{
                        console.log(info);
                    });
                }}/>
            </View>
        );
    }
}
```

抽 丝 剥 茧

若要在 Android 平台获取用户网络环境信息，需要在 AndroidManifest.xml 文件中添加如下代码来获取权限：

```
    <uses-permission android:name="android.permission.
ACCESS_NETWORK_STATE" />
```

NetInfo 对象中的 fetch 方法用异步的方式获取用户所处的网络环境，这个方法会返回一个 Promise 对象，在其完成的回调中会将用户的网络状态作为参数传入。注意，在 iOS 平台和 Android 平台所定义的网络环境并不一致。

（1）iOS 平台

- none：设备处于离线状态。
- wifi：设备处理 wifi 联网状态。
- cell：设备通过移动网络连接。
- unknown：未知状态。

（2）Android 平台

- NONE：设备处于离线状态。
- BLUETOOTH：蓝牙数据连接。
- DUMMY：模拟数据连接。
- MOBILE：移动网络数据连接。
- MOBILE_DUN：拨号网络连接。
- MOBILE_HIPRI：高优先级移动网络连接。
- MOBILE_MMS：彩信移动网络连接。
- MOBILE_SUPL：安全用户面定位（SUPL）数据连接。
- VPN：虚拟网络连接。

- WIFI：wifi 连接。
- WIMAX：wimax 连接。
- UNKNOWN：位置状态。

你也可以使用 NetInfo 对象的 addEventListener 方法来添加用户网络环境变化的监听，这个方法需要两个参数，第 1 个参数为监听的行为，目前只能填写字符串“change”，第 2 个参数为回调函数，当用户网络环境改变时，这个回调函数会被调用，并将当前的网络环境作为参数传入。与 addEventListener 方法对应，removeEventListener 方法用来移除监听。

对于 Android 平台，NetInfo 中还提供了一个更方便的方法获取用户所在的网络连接是否计费（大多数时候这个方法更常用）：isConnectionExpensive。这个方法需要传入一个函数作为回调，函数的参数为布尔值，表示用户当前的网络环境是否计费。

还有一点需要注意，无论是在 iOS 平台还是 Android 平台，NetInfo 对象中都提供了 isConnencted 属性，这个属性也是一个对象，你也可以使用 isConencted 对象来调用 fetch、addEventListener 和 removeEventListener 方法。不同的是，使用这个对象在调用上面的方法时，只会将用户当前是否联网的信息传入，在你不关心具体的网络环境类型时这个对象十分好用。

8.17　React Native 动画技术

动画的合理应用可以大大地增强应用程序的用户体验。在 React Native 中提供了强大的动画渲染框架 Animated。Animated 模块中提供了大量的动画对象类与动画执行方法。要想让一个动画效果完整执行，你至少需要关注两部分内容：对话对象的创建与动画执行函数。

需要注意，并不是所有的组件默认都支持动画，React Native 中默认支持的动画组件有 View、Text、Image 和 ScrollView，当然你也可以根据自己的需要定义自定义的动画组件，后面章节会更加深入地介绍。

8.17.1　创建单值驱动的动画

单值驱动是指通过不断更新动画组件某个属性的数值来实现动画效果，例如控件的宽高变化动画、组件的位置变化动画、组件的透明度变化动画等。单值驱动动画使用 AnimatedValue 类进行配置，AnmatedValue 是 Animated 框架中的内部类，在使用时，使用 Animated.Value()的方式创建。

AnimatedValue 主要通过定义起始值与结束值来定义动画过程，在 Animated.Value()构造方法中需要传入一个数值，作为定义动画属性的起始值。其他重要方法如表 8-27 所示。

表 8-27　AnimatedValue 的方法

方法名	解释	平台	参数
setValue	重新定义动画起始值，执行这个方法会中断当前动画	通用	数值
setOffset	设置修正值，之后使用 setValue 方法设置的值会被加上这个修正值	通用	数值

（续表）

方法名	解释	平台	参数
addListener	添加一个动画值变化的监听，会返回一个监听 id 字符串	通用	函数，会将当前动画执行过程中属性值包装成对象作为参数进行传入
removeListener	根据 id 移除一个监听	通用	字符串
stopAnimation	立即结束动画	通用	函数，函数的参数为动画完全完成后的最终属性值
resetAnimation	立即结束动画，并且将动画属性值设置为初始状态	通用	函数，函数的参数为动画完全完成后的最终属性值

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 AnimatedDemo.js 的文件，在其中编写如下测试代码：

```
import React, { Component } from 'react';
import {
  Button,
  View,
  Animated
} from 'react-native';
export default class AnimatedValueDemo extends Component{
    constructor(props){
        super(props);
        this.state = {
            animated:new Animated.Value(1)
        };
        this.state.animated.addListener((value)=>{
            console.log(value);
        });
    }
    render(){
        return(
            <View>
                <Animated.View style={{marginTop:100,height:
this.state.animated,backgroundColor:'red'}}></Animated.View>
                <Button title="Start Animation" onPress={()=>{
                    Animated.timing(
                        this.state.animated,
                        {toValue:100}
                    ).start();
                }}/>
            </View>
        );
    }
}
```

在上面代码创建 AnimatedValue 对象的部分，先把这个对象作为 AnimatedValueDemo 组件的状态进行保存，之后将其与 Animated.View 组件的高度属性进行绑定（要执行动画，必须使用支持动画的组件）。初始状态时，动画视图的高度为 1 个单位，单击按钮，动画视图的高度调整为 100 个单位，接下来再来分析动画的执行方法 timing 函数。

8.17.2　使用 timing 方法执行平滑过渡动画

timing 函数是 Animated 框架中提供的一个执行 AnimatedVlaue 或者 AnimatedValueXY（二维值驱动的动画，后面会介绍）类型动画的方法。这个方法可以传入两个参数，第一个参数为要执行的动画对象，第 2 个参数为一些动画选项配置对象。我们主要来看第 2 个参数，这个参数中可配置的属性如表 8-28 所示。

表 8-28　timing 函数参数中可配置的属性

属性名	解释	平台	值类型或可选值
isInteraction	动画执行时是否可进行用户交互（例如动画执行时触发 onPress 方法）	通用	布尔值
useNativeDriver	是否进行原生驱动	通用	布尔值
onComplete	动画完成后执行的回调	通用	函数，会将动画是否完整执行作为布尔值包装成对象传入
iterations	迭代次数	通用	数值
toValue	动画属性目标值	通用	数值或 AnimatedValue 对象
duration	动画执行时间	通用	数值
delay	动画延时执行实现	通用	数值
easing	动画执行时间函数，Easing 模块中定义了许多函数，你也可以定义自己的时间函数	通用	函数

timing 返回一个动画组件，调用 start 方法即可开始动画的执行。修改 AnimatedValueDemo 组件的代码如下：

```
    export default class AnimatedValueDemo extends Component{
        constructor(props){
            super(props);
            this.state = {
                animated:new Animated.Value(1)
            };
            this.state.animated.addListener((value)=>{
                console.log(value);
            });
        }
        render(){
            return(
                <View>
                    <Animated.View style={{marginTop:100,
height:this.state.animated,backgroundColor:'red'}}></Animated.View>
```

```
                    <Button title="Start Animation" onPress={()=>{
                        Animated.timing(
                            this.state.animated,
                            {
                                toValue:100,
                                onComplete:(obj)=>{
                                    console.log(obj);
                                },
                                duration:2,
                                delay:1
                            }
                        ).start();
                    }}/>
                </View>
            );
        }
    }
```

修改 index 相关文件后，运行工程，可以看到动画更加可控的效果。

8.17.3 深入理解 easing

easing 函数用来控制动画受时间影响的方式，例如大部分动画的执行是线性的，即动画是匀速平稳执行的，还有些动画的执行则是先快后慢，或者先慢后快，再或者开始结束慢、中间快。要配置这样的动画，就需要使用 easing 函数。

在 7.17.2 小节介绍动画的配置对象时，你知道可以使用 easing 属性来配置动画。这个属性需要设置为函数，函数中会将动画当前执行的进度作为参数（0～1）传入，需要开发者将动画实际的执行程度作为返回值返回，举一个简单的例子，我们前面配置的动画是在 2 秒内，View 组件的高度由 1 个单位变化成 100 个单位，你可以将这个变化尺寸通过 easing 属性来变成 200 个单位，示例代码如下：

```
    <View>
                    <Animated.View style={{marginTop:100, height:
this.state.animated,backgroundColor:'red'}}></Animated.View>
                    <Button title="Start Animation" onPress={()=>{
                        Animated.timing(
                            this.state.animated,
                            {
                                toValue:100,
                                onComplete:(obj)=>{
                                    console.log(obj);
                                },
                                duration:2000,
                                delay:1000,
                                easing:(oriValue)=>{
```

```
                                return 2*oriValue;
                            }
                        }
                    ).start();
                }}/>
    </View>
```

一般情况下，你都不需要手动创建 easing 函数（除非真的需要）。React Native 中提供了 Easing 模块，这个模块中提供了大量的函数供开发者直接使用，如表 8-29 所示。

表 8-29　Easing 模块的函数

函数名	解释	平台	参数
step0	符号函数，当传入值大于 0 时返回 1，否则返回 0	通用	数值
step1	符号函数，当传入值大于 1 时返回 1，否则返回 0	通用	数值
linear	线性函数，直接将参数作为返回值返回	通用	数值
ease	缓慢加速函数	通用	数值
quad	平方函数，将参数求平方后返回	通用	数值
cubic	立方函数，将参数求立方后返回	通用	数值

poly	指数函数构造器，此函数的返回值不是数值，而是一个 easing 函数，此 easing 函数会将 poly 函数的参数作为指数，将自身的参数作为底数进行运算后返回，例如 poly(3)会返回一个函数：
	(t)=>{
	　return Math.pow(t,3);
	}
		通用	数值

sin	sin 函数	通用	数值
circle	圆函数	通用	数值
exp	指数函数	通用	数值
elastic	弹性函数构造器，传入弹性系数，会返回一个 easing 函数	通用	数值
back	后略函数构造器，返回 easing 函数	通用	数值
bounce	回弹函数，传入弹性系统	通用	数值
bezier	贝塞尔函数构造器，传入 4 个贝塞尔参数值，返回 easing 函数	通用	(x1,y1,x2,y2)
in	顺序 easing 函数，传入 easing 函数，并且直接返回	通用	easing 函数
out	逆序 easing 函数，传入 easing 函数，逆序整理后返回	通用	easing 函数
inOut	传入 easing 函数，前半部分顺序，后半部分逆序整理后返回	通用	easing 函数

完整的回弹动画代码如下：

```
import React, { Component } from 'react';
import {
  Button,
  View,
```

```
        Animated,
        Easing
    } from 'react-native';
    export default class AnimatedDemo extends Component{
        constructor(props){
            super(props);
            this.state = {
                animated:new Animated.Value(1)
            };
            this.state.animated.addListener((value)=>{
                console.log(value);
            });
        }
        render(){
            return(
                <View>
                    <Animated.View style={{marginTop:100,height:this.state.animated,
backgroundColor:'red'}}></Animated.View>
                    <Button title="Start Animation" onPress={()=>{
                        Animated.timing(
                            this.state.animated,
                            {
                                toValue:100,
                                onComplete:(obj)=>{
                                    console.log(obj);
                                },
                                duration:2000,
                                delay:1000,
                                easing:Easing.bounce
                            }
                        ).start();
                    }}/>
                </View>
            );
        }
```

8.17.4 二维动画对象与衰减动画

如果你认真学习了前 3 小节的内容，就应该对 React Native 中动画的构建有了全局的理解，再学习后面的动画技术将会轻松很多。本小节将介绍二维动画与 decay 动画函数。

如果你想将组件的某一个属性进行动画变换，那么使用 AnimatedValue 是十分有效的。React Native 中还提供了另一个动画对象 AnimatedValueXY，它可以方便地控制组件的两个属性（我们姑且将其称为二维动画），使用 Animated.ValueXY 来进行对象的创建，原则上讲，也可以通过创建两个 AnimatedValue 来实现二维动画。timing 方法可以方便地控制动画执行的方式，Easing 函数的

使用更是增强的动画地表现形式，decay 方法也是用来执行动画的函数，只是它没有 timing 方法灵活，用来执行随时间进行衰减的动画（减速效果的动画）。

在 AnimatedDemo.js 文件中创建一个新类，代码如下：

```
export class AnimatedValue2DDemo extends Component{
    constructor(props){
        super(props);
        this.state = {
            animated:new Animated.ValueXY({x:10,y:10})
        };
        this.state.animated.addListener((value)=>{
            console.log(value);
        });
    }
    render(){
    return(
        <View>
            <Animated.View style={{marginTop:100,height:this.state.animated.x,
width:this.state.animated.y,backgroundColor:'red'}}></Animated.View>
                <Button title="Start Animation" onPress={()=>{
                    Animated.decay(
                        this.state.animated,
                        {
                        velocity:{x:0.3,y:0.3},
                        deceleration:0.997
                        }
                    ).start();
                }}/>
        </View>
        );
    }
}
```

修改 index.ios.js 与 index.android.js 文件如下：

```
import AnimatedDemo,{AnimatedValue2DDemo} from './Demo/AnimatedDemo';
export default class RN extends Component {
  render() {
    return (
      <AnimatedValue2DDemo />
    );
  }
}
```

运行工程，可以明显地看到在组件尺寸增大的过程中，增大的速度不断衰减。注意，decay 方法的第 2 个参数用来进行动画配置，其中 velocity 属性用来配置动画执行的初速度，deceleration 属性用来设置衰减速率。

8.17.5 弹簧动画

decay 方法所执行的动画是持续衰减的减速动画，Animated 框架中还提供了一个 spring 方法，专门用来执行弹簧动画。

在 AnimatedDemo.js 文件中新建一个类，代码如下：

```
    export class SpringAnimatedDemo extends Component{
        constructor(props){
            super(props);
            this.state = {
                animated:new Animated.ValueXY({x:10,y:10})
            };
        }
        render(){
        return(
            <View>
                <Animated.View style={{marginTop:100,height:this.state.animated.x,
width:this.state.animated.y,backgroundColor:'red'}}></Animated.View>
                    <Button title="Start Animation" onPress={()=>{
                        Animated.spring(this.state.animated,{
                            toValue:{x:100,y:100},
                            onComplete:()=>{
                                console.log("complete");
                            },
                            bounciness:10
                            }).start();
                    }}/>
            </View>
            );
        }
    }
```

修改 index.ios.js 与 index.android.js 文件后，运行工程，可以看到组件尺寸变大的过程中会出现回弹效果。

抽 丝 剥 茧
使用 timing 方法也可以实现弹簧动画，只是 spring 方法的动画配置参数中提供了更方便的弹簧系数设置方式。

spring 方法的动画配置对象中的可设置属性如表 8-30 所示。

表 8-30　spring 方法动画配置对象中的可设置属性

属性名	解释	平台	值类型或可选值
toValue	设置动画结束值	通用	数值或对应动画对象
overshootClamping	设置是否限制边界	通用	布尔值
restDisplacementThreshold	位移阈值	通用	数值
restSpeedThreshold	速度阈值	通用	数值
velocity	设置初速度	通用	数值
bounciness	设置弹力	通用	数值
speed	设置速度	通用	数值
tension	设置张力	通用	数值
friction	设置摩擦力	通用	数值

8.17.6　Interpolation 插值动画

在实际开发中使用的动画效果往往不是单独动作，而是一系列动作。想象一下，你现在需要将一个视图的尺寸先放大，之后再缩小，该怎么做呢？当然你可以在放大动画结束后再缩小动画，只是使用 Interpolator 插值动画不仅可以省掉冗余代码，还会使代码更加优雅。

Interpolation 动画的原理是进行值的映射，例如动画视图高度从 10 单位修改成 100 单位，就可以使用插值动画将其分成两个部分，当值从 10 到 50 之间递增时进行视图尺寸的放大，当值从 50 到 100 之间递增时进行视图尺寸的缩小。在 AnimatedDemo.js 文件中创建一个新类，代码如下：

```
    export class InterpolationAnimatedDemo extends Component{
        constructor(props){
            super(props);
            this.state = {
                animated:new Animated.Value(10),
            };
            this.bindAnimated = new Animated.Interpolation(this.state.animated,{
                        inputRange:[10,50,100],
                        outputRange:[10,100,50],
                        extrapolate:'identity'
                    })
        }
        render(){
        return(
            <View>
                <Animated.View style={{marginTop:100,height:this.bindAnimated,
backgroundColor:'red'}}></Animated.View>
                    <Button title="Start Animation" onPress={()=>{
                        Animated.timing(this.state.animated,{
                            toValue:100,
                            duration:2000
```

```
                        }).start();
                }}/>
        </View>
        );
    }
}
```

简单理解，Interpolation 的作用是对 AnimatedValue 对象进行包装，当 AnimatedValue 动画在执行时，其值会进行映射后再作用于视图上，你可以任意设置插值个数，inputRange 设置输入值节点，outputRange 设置输出值节点，其中还可以利用 extrapolate 属性来设置当输入值超出边界时的处理方式，其可选设置值为“extend”“identity”或“clamp”，extend 表示允许输入值超出边界，clamp 表示不允许输入值超出边界，identity 表示当输入值超出边界后返回输入值本身。你也可以单独设置 extrapolateLeft 或 extrapolateRight 属性来设置左侧输入值超边界或右侧输入值超边界的处理情况。

抽 丝 剥 茧
extrapolateLeft 还可以将输入值映射成弧度或角度，在旋转动画中十分有用。

8.17.7 聚合动画值

Animated 模块中提供了许多聚合动画值的方法，可以将 AnimatedValue 动画的值进行聚合计算后再作用于视图组件上，请看如下示例代码：

```
export class AnimatedCalculateDemo extends Component{
    constructor(props){
        super(props);
        var animated1 = this.animated1 = new Animated.Value(10);
        var animated2 = this.animated2 = new Animated.Value(100);
        this.addAnimated = Animated.add(animated1,animated2);
    }
    render(){
    return(
        <View>
            <Animated.View style={{marginTop:100,height:this.addAnimated,
backgroundColor:'red'}}></Animated.View>
                <Button title="Start Animation" onPress={()=>{
                    Animated.timing(this.animated1,{
                        toValue:100,
                        duration:2000
                        }).start();
                }}/>
        </View>
        );
    }
}
```

add 方法的作用是将两个动画的值相加作为视图最终渲染的值，被聚合的两个动画既可以单独执行也可以同时执行，与 add 方法相似，Animated 中支持 4 种聚合动画的方式，如表 8-31 所示。

表 8-31　Animated 支持的 4 种聚合动画方式

方法名	解释	平台	参数
add	将两个动画的值进行相加计算	通用	(a,b) • a：AnimatedValue 动画对象或数值 • b：AnimatedValue 动画对象或数值
divide	将两个动画的值进行相除计算	通用	(a,b) • a：AnimatedValue 动画对象或数值 • b：AnimatedValue 动画对象或数值
multiply	将两个动画的值进行相乘计算	通用	(a,b) • a：AnimatedValue 动画对象或数值 • b：AnimatedValue 动画对象或数值
modulo	将动画的值进行取模计算	通用	(a,mo) • a：AnimatedValue 动画对象 • mo：取模数值

抽 丝 剥 茧

除了上面列出的 4 个方法，Animated 中还提供了一个 diffClamp 方法，这个方法的使用和上面 4 个方法相似，用来限制动画的值在某个范围之内：

方法名	解释	平台	参数
diffClamp	限制动画的值	通用	(animated,min,max) • animated：动画对象 • min：最小值 • max：最大值

8.17.8　组合动画

将动画进行组合往往会产生更加炫酷的效果。Animated 框架中提供了链式组合动画 sequence 方法、并行组合动画 parallel 方法和重叠组合动画 stagger 方法，如表 8-32 所示。

表 8-32　Animated 框架的组合动画方法

方法名	解释	平台	参数
sequence	依次执行一组动画动作	通用	动画动作数组

parallel	同时执行一组动画动作	通用	(array,config)
			array：动画动作数组
			config：动画配置对象，其中可设置属性如下：
			{
			stopTogether：布尔值，当某个动画停止时，是否其他动画也随着停止
			}

（续表）

方法名	解释	平台	参数
stagger	重叠执行一组动画动作，通过延时间隔	通用	(number,array) • number：延时时间 • array：动画动作数组

sequence 方法会无缝地执行一组动画。如果需要在动画间插入一些延时，可以使用 Animated 中的 delay 方法，这个方法可以传入一个数值作为延时参数，并返回一个空的动画动作来消耗时间。示例代码如下：

```
    export class AnimatedGroupeDemo extends Component{
        constructor(props){
            super(props);
            this.animated1 = new Animated.Value(10);
            this.animated2 = new Animated.Value(10);
            this.animated3 = new Animated.Value(10);
        }
        render(){
        return(
            <View>
                <Animated.View style={{marginTop:100,height:30,
width:this.animated1,backgroundColor:'red'}}></Animated.View>
                <Animated.View style={{marginTop:10,height:30,
width:this.animated2,backgroundColor:'red'}}></Animated.View>
                <Animated.View style={{marginTop:10,height:30,
width:this.animated3,backgroundColor:'red'}}></Animated.View>
                    <Button title="Start Sequence Animation" onPress={()=>{
                        var a1 = Animated.timing(this.animated1,{
                            toValue:100,
                            duration:2000
                            });
                        var a2 = Animated.timing(this.animated2,{
                            toValue:100,
                            duration:2000
                            });
                        var a3 = Animated.timing(this.animated3,{
                            toValue:100,
                            duration:2000
                            });
                        Animated.sequence([a1,a2,Animated.delay(2000),
a3]).start();
                    }}/>
                    <Button title="Start Parallel Animation" onPress={()=>{
                        var a1 = Animated.timing(this.animated1,{
                            toValue:100,
```

```
                        duration:2000
                        });
                    var a2 = Animated.timing(this.animated2,{
                        toValue:100,
                        duration:2000
                        });
                    var a3 = Animated.timing(this.animated3,{
                        toValue:100,
                        duration:2000
                        });
                    Animated.parallel([a1,a2,a3],{stopTogether:
false}).start();
                }}/>
                <Button title="Start Stagger Animation" onPress={()=>{
                    var a1 = Animated.timing(this.animated1,{
                        toValue:100,
                        duration:2000
                        });
                    var a2 = Animated.timing(this.animated2,{
                        toValue:100,
                        duration:2000
                        });
                    var a3 = Animated.timing(this.animated3,{
                        toValue:100,
                        duration:2000
                        });
                    Animated.stagger(500,[a1,a2,a3]).start();
                }}/>
            </View>
            );
        }
    }
```

注意，虽然上面的代码演示了在多个组件上执行组合动画，实际上你也可以在同一个视图组件的不同属性上执行组合动画。

8.17.9 循环动画

使用 Animated 做循环动画也十分容易，Animated 框架中提供了 loop 方法，这个方法接收两个参数，第 1 个参数是要执行的动画动作，第 2 个参数为循环动画配置对象，其中 iterations 属性用来设置循环次数，示例如下：

```
        constructor(props){
            super(props);
            this.animated1 = new Animated.Value(10);
        }
```

```
        render(){
        return(
            <View>
                <Animated.View style={{marginTop:100,height:30,
width:this.animated1,backgroundColor:'red'}}></Animated.View>
                    <Button title="Start Loop Animation" onPress={()=>{
                        var a1 = Animated.timing(this.animated1,{
                            toValue:100,
                            duration:2000
                            });
                        Animated.loop(a1,{iterations:-1}).start();
                    }}/>
            </View>
            );
        }
```

注意，iterrations 属性如果设置为 0，则动画不会执行，设置为-1 则会无限循环执行。

抽 丝 剥 茧
Animated 的 loop 方法实际上返回的也是动画动作对象，可以结合组合动画使用。

8.17.10 布局动画

前面学习的动画技术足够强大，但其并不总是十分方便。Animated 相关动画作用于一个组件上是非常好用的，但是如果在动画的进行中多个组件会受影响，那么使用 React Native 中的 LayoutAnimation 则会更加容易。

LayoutAnimation 是专门提供给布局使用的动画对象，当界面布局发生改变时，使用它可以产生动画效果，包括组件位置的变化、组件尺寸的变化以及组件添加或删除等。在 AnimatedDemo.js 文件中添加如下示例代码：

```
    import {
      UIManager
    } from 'react-native';
    UIManager.setLayoutAnimationEnabledExperimental &&
      UIManager.setLayoutAnimationEnabledExperimental(true);
    export class LayoutAnimationDemo extends Component{
        constructor(props){
            super(props);
            this.state={
                width:100,
                height:100
            }
        }
        render(){
        return(
```

```
            <View>
                <View style={{marginTop:100,height:this.state.height,
width:this.state.width,backgroundColor:'red'}}></View>
                    <Button title="Start Animation" onPress={()=>{
                        LayoutAnimation.spring();
                        this.setState({
                            width:this.state.width+10,
                            height:this.state.height+10
                            });
                    }}/>
            </View>
            );
        }
    }
```

上面的 UIManager 相关代码是为了在 Android 平台开启布局动画所需要的。LayoutAnimation 的使用十分简单，只需要在布局修改之前调用如表 8-33 所示的 3 个方法中的一个即可。

表8-33　LayoutAnimation调用方法

方法名	解释	平台	参数
easeInEaseOut	使用淡入淡出动画	通用	无
linear	使用线性动画	通用	无
spring	使用弹簧动画	通用	无

8.17.11　自定义组件动画

React Native 默认支持动画的组件只有 Image、Text、View 与 ScrollView。在实际开发中，大部分视图组件都是我们自定义的组件，要使自定义的组件可以支持动画，也十分简单，只需要使用 createAnimatedComponent 方法进行包装即可。

在 HelloWorld 工程 Demo 文件夹下新建一个命名为 CustomAnimatedView.js 的文件，将其作为自定义组件，在其中编写如下代码：

```
    import React, { Component } from 'react';
    import {
      View,
      Text
    } from 'react-native';
    export default class CustomAnimatedView extends Component{
        render(){
            return(
                <View>
                    <Text style={{marginTop:100,textAlign:'center',fontSize:24,
opacity:this.props.opacity}}>自定义组件</Text>
```

```
            </View>
        );
    };
}
```

在 AnimatedDemo.js 文件中新建一个类，用于测试动画，代码如下：

```
export class CustomAnimationViewDemo extends Component{
    constructor(props){
        super(props);
        this.animated1 = new Animated.Value(1);
        this.aniView = Animated.createAnimatedComponent(CustomAnimatedView);
    }
    render(){
        return(
            <View>
                <this.aniView opacity={this.animated1}/>
                <Button title="Start Animation" onPress={()=>{
                    Animated.timing(this.animated1,{
                        toValue:0,
                        duration:2000
                        }).start();
                }}/>
            </View>

        );
    }
}
```

修改 index 相关文件后，运行工程，可以看到自定义的组件也可以支持动画效果。

8.18　调用设备振动模块

振动作为移动设备上的一种触感反馈是一种十分重要的交互方式。在 React Native 中调用设备的振动模块也十分容易。React Native 中提供了一个 Vibration 对象，此对象中封装了两个方法，如表 8-34 所示。

注意，在 Android 平台上调用振动模块需要在 AndroidManifest.xml 文件中添加如下权限申请代码：

```
<uses-permission android:name="android.permission.VIBRATE"/>
```

表 8-34　Vibration 对象的方法

方法名	解释	平台	参数
vibrate	此方法用来调用设备的振动模块	通用	接收两个参数(array,re) • array：数值数组，在 Android 平台，第 1 个元素表示延时时间，第 2 个元素表示振动时长，依此类推，第 3 个参数为延时，第 4 个参数为振动时长。在 iOS 平台则不同，iOS 设备的振动时长固定，这个数组中所有的参数均表示延时时长 • re：表示是否循环振动，如果设置为 YES，则只能通过调用 cancel 方法结束振动
cancel	结束振动	通用	无

示例代码如下：

```
import React, { Component } from 'react';
import {
  Button,
  View,
  Vibration
} from 'react-native';
export default class VibrationDemo extends Component{
    render(){
        return(
            <View>
                <Button title="Vibration" onPress={()=>{
                    Vibration.vibrate([1000,2000,1000,2000],false);
                }}/>
            </View>
        );
    }
}
```

博 闻 强 识

注意，本节内容需要在真机上验证，关于如何将 React Native 项目运行在真机上，后面会有专门的介绍。

8.19　封装滑动手势

你应该还记得，我们在学习 View 组件的时候曾介绍过 React Native 组件的用户触摸系统，View 组件中提供了许多回调属性来响应用户的触摸请求。在实际开发中，除了点按手势外，最常用的要数滑动手势了。React Native 中提供了 PanResponder 对象来更轻松地创建滑动手势。

PanResponder 对象调用 create 方法可以创建滑动手势对象，其中可以设置许多回调属性，如表 8-35 所示。

表 8-35　PanResponder 可以设置的回调属性

属性名	意义	平台	值类型
onMoveShouldSetPanResponder	设置是否允许成为移动响应接收者	通用	函数，需要返回布尔值
onMoveShouldSetPanResponderCapture	设置是否允许拦截子组件移动响应	通用	函数，需要返回布尔值
onStartShouldSetPanResponder	设置是否允许成为触摸事件响应接收者	通用	函数，需要返回布尔值
onStartShouldSetPanResponderCapture	设置是否允许拦截子组件触摸事件响应	通用	函数，需要返回布尔值
onPanResponderGrant	成为响应者回调的函数	通用	函数
onPanResponderStart	开始响应触摸事件回调	通用	函数
onPanResponderReject	拒绝成为事件响应者回调	通用	函数
onPanResponderEnd	完成事件响应回调	通用	函数
onPanResponderRelease	触摸结束回调	通用	函数
onPanResponderMove	触摸点移动时回调	通用	函数
onPanResponderTerminate	触摸事件被中断时回调	通用	函数
onPanResponderTerminationRequest	接收到触摸事件中断请求时回调	通用	函数

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 PanResponderDemo.js 的文件，在其中编写如下代码：

```
import React, {
    Component
} from 'react';
import {
    View,
    PanResponder,
    Text
} from 'react-native';
export default class PanResponderDemo extends Component {
    constructor(props) {
        super(props);
        this.state = {
            eventName: '',
            pos: '',
        };
        this.myPanResponder = PanResponder.create({
            //要求成为响应者:
            onStartShouldSetPanResponder: (evt, gestureState) => true,
            onStartShouldSetPanResponderCapture: (evt, gestureState) => true,
```

```
                onMoveShouldSetPanResponder: (evt, gestureState) => true,
                onMoveShouldSetPanResponderCapture: (evt, gestureState) => true,
                onPanResponderTerminationRequest: (evt, gestureState) => true,
                //响应对应事件后的处理:
                onPanResponderGrant: (evt, gestureState) => {
                    this.state.eventName = '触摸开始';
                    this.forceUpdate();
                },
                onPanResponderMove: (evt, gestureState) => {
                    var _pos = 'x:' + gestureState.moveX + ',y:' +
gestureState.moveY;
                    this.setState({
                        eventName: '移动',
                        pos: _pos
                    });
                },
                onPanResponderRelease: (evt, gestureState) => {
                    this.setState({
                        eventName: '抬手'
                    });
                },
                onPanResponderTerminate: (evt, gestureState) => {
                    this.setState({
                        eventName: '另一个组件已经成为了新的响应者'
                    })
                },
            });
        }
        render() {
            return (
            <View style={{flex:1}} {...this.myPanResponder.panHandlers}>
            <Text
style={{top:100,textAlign:'center'}}>eventName:{this.state.eventName}|{this.st
ate.pos}</Text>
            </View>);
        }
```

运行工程，在屏幕上移动手指，可以看到 Text 组件显示出的手指位置信息。

抽 丝 剥 茧
滑动手势对象的 panHandlers 属性是对 View 触摸事件的集合包装对象。上面的代码使用了一个技巧，“...” 运算符会将对象的属性进行拆解，作为 View 组件的属性。

8.20 获取屏幕尺寸信息

在实际开发中，时常需要获取设备的屏幕尺寸信息，这对布局十分重要。在 React Native 中提供了 Dimensions 对象来提取设备屏幕尺寸信息。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 DimensionsDemo.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  View,
  Text,
  Dimensions
} from 'react-native';

export default class DimensionsDemo extends Component{
    render(){
        return(
            <View style={{paddingTop:100}}>
                <Text>{"屏幕尺寸信息"+Dimensions.get('window').width+"-"+
Dimensions.get('window').height}</Text>
            </View>
        );
    }
}
```

get 方法用来获取设备屏幕尺寸信息，运行工程，效果如图 8-25 所示。

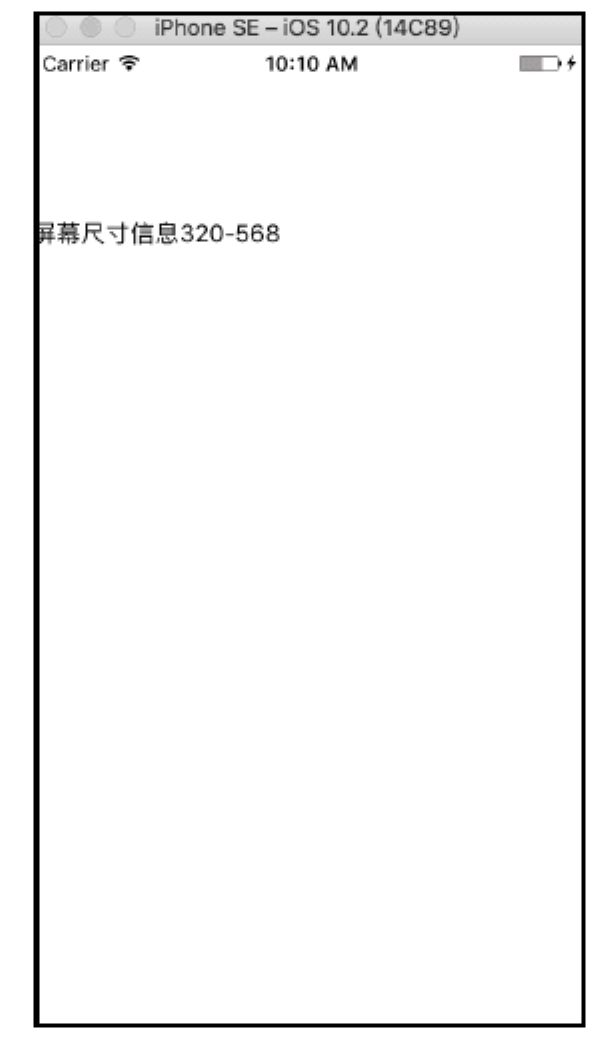


图 8-25 获取设备屏幕尺寸信息

8.21　特定平台代码

虽然 React Native 提供了基本统一的跨平台开发接口，但是针对不同的平台，例如在 Android 和 iOS 上，我们还是希望可以根据用户的习惯进行分平台定制。其实前面你已经学习过，可以通过在文件命名时添加 ios 或者 android 后缀来使 React Native 分平台加载文件，在同一个文件中，你也可以分平台来编写代码。

React Native 中提供了 Platform 对象，这个对象中包含有平台信息，并且提供了简洁的方法来支持分平台代码编写。Platform 对象中提供的属性如表 8-36 所示。

表 8-36　Platform 对象中提供的属性

属性名	意义	平台	值类型或可选值
OS	获取当前系统平台	通用	字符串 "ios"或"android"
Version	获取当前系统版本	通用	字符串
isPad	设备是否是 iPad	iOS	布尔值
isTVOS	是否是 TVOS 系统	iOS	布尔值

Platform 对象中还提供了一个 select 函数，这个函数需要接收一个对象作为参数，对象中可以定义 3 个属性，即 ios、android 和 default。 当系统为 iOS 时，select 方法会将其中 ios 属性对应的值返回；当系统是 Android 时，select 方法会将属性 android 对应的值返回；default 属性用于设置默认值，当没有设置 ios 属性或 android 属性时会返回 default 属性的值。

在 HelloWorld 工程的 Demo 文件夹中新建一个命名为 PlatformDemo.js 的文件，在其中编写如下示例代码：

```
import React, { Component } from 'react';
import {
  View,
  Text,
  Platform,
  StyleSheet
} from 'react-native';

export default class PlatformDemo extends Component{
    constructor(props){
        super(props);
        if (Platform.OS==="ios") {
            this.title = "ios";
        }
        if (Platform.OS==="android"){
            this.title = "android";
```

```
            }
            console.log(Platform.Version);
        }

        render(){
            return(
                <View >
                    <Text  style={sty.view}>{this.title}</Text>
                </View>
            );
        }
    }

    var sty = StyleSheet.create({
        view:{
            ...Platform.select({
                default:{backgroundColor:'red',flex:1},
                ios:{marginTop:100,color:'blue',fontSize:24,textAlign:
'center'},
                android:{marginTop:200,color:'green',fontSize:24,textAlign:
'center'}
            })
        }
    });
```

分别在 iOS 和 Android 平台运行工程，可以看到不同平台的不同效果。

8.22　定时器的简单应用

在原生应用开发中，常常会使用到定时器。使用定时器的场景无非两种，一种是延迟一定时间后执行某个事件，一种是每间隔一定时间执行某个事件。React Native 中提供了与 Web 开发中一致的定时器方法，如表 8-37 所示。

表 8-37　Native 提供的定时器方法

方法名	意义	平台	参数
setTimeout	延时一定时间后执行回调函数	通用	(func,time,...) • func：回调函数 • time：延时时间，单位为毫秒 • ...：可以传入回调函数的参数
setInterval	设置每间隔一定时间执行回调函数	通用	(func,time) • func：回调函数 • time：间隔时间，单位为毫秒

React Native 中还提供了 clearTimeout 与 clearInterval 函数，使用这两个方法可以手动停止定时器。有一点需要格外注意，当组件卸载时，必须将组件中的定时器清除掉，否则会出现意外的崩溃现象。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 TimerDemo.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  Button,
  View
} from 'react-native';

export default class TimerDemo extends Component{
    render(){
        return(
            <View style={{marginTop:100}}>
                <Button title="TimeOut" onPress={()=>{
                    this.timerout=setTimeout(()=>{
                        console.log("TimeOut");
                    }, 3000);
                }}/>
                <Button title="TimeInterval" onPress={()=>{
                    this.timeInterval=setInterval(()=>{
                        console.log("setInterval");
                        },3000);
                }}/>
            </View>
        );
    }
    componentWillUnmount(){
        this.timerout&&clearTimeout(this.timerout);
        this.timeInterval&&clearInterval(this.timeInterval);
    }
}
```

运行工程，打开调试模式，可以看到定时器的执行状况。定时器时常用于按照一定频率刷新界面，例如有倒计时模块的界面，可以使用定时器来进行界面时间 Text 标签的刷新。

第 9 章

实战项目：汇率转换器

从本章开始，介绍几个真实项目的训练，你将使用前边所学习的各种知识开发出一些属于你的完整应用程序。完整项目的开发学习和独立组件或独立技术的学习有很大不同，在学习独立组件时，你需要关注组件如何用和怎么用，而在学习完整应用时，你需要将更多的精力放在思考程序的逻辑、设计程序的结构上。不过不用担心，在后面几章的学习中，我们将一起由简入深、由易到难，逐步掌握 React Native 开发技术。

本章将从最基础的单界面应用程序开始，汇率转化器是一款用于汇率计算的小工具软件，支持人民币与美元的互相转换，完整应用效果如图 9-1 所示。

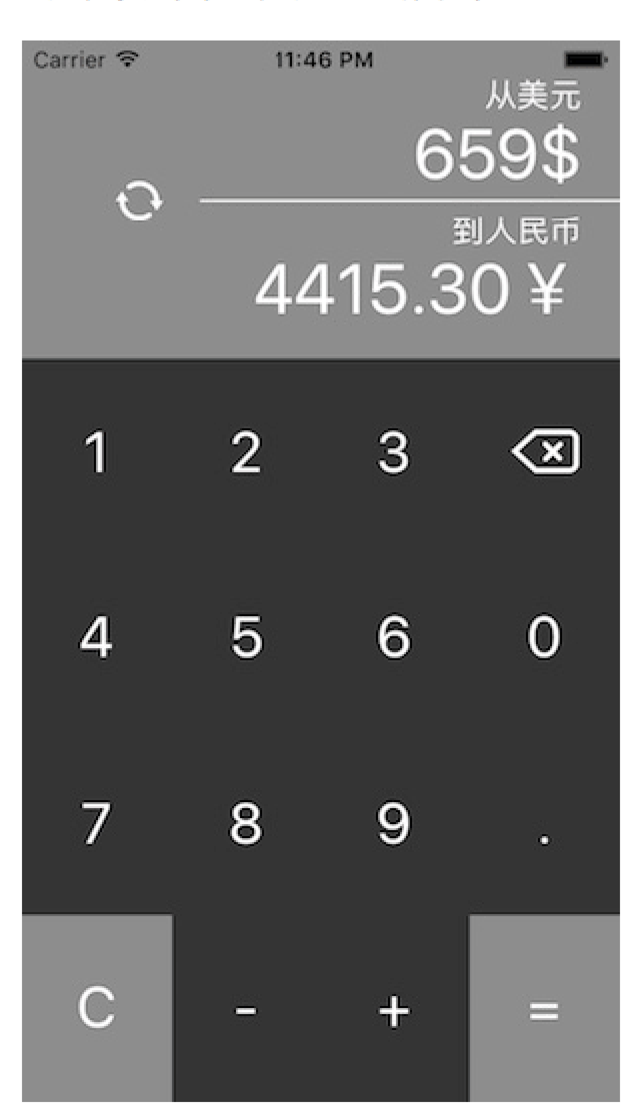


图 9-1　汇率转化器

下面我们就来一步步完成第一款 React Native 应用。

9.1　搭建汇率转换器项目主界面

在开始应用程序开发之前，首先需要确定程序的结构。清晰的结构可以使应用程序的维护与扩展变得更加容易。首先打开终端，在应用目录下新建一个 React Native 项目，使用如下命令：

```
react-native init Calu
```

Calu 为汇率转换器项目的工程名。使用 Sublime Test 工具打开工程，在根目录下创建命名为 ViewController 和 View 的文件夹，分别放置项目的视图控制器文件与视图文件。在进行项目结构的设计时，建议从最基础的 MVC 模式开始，这也是本书实战项目所选用的结构。在 MVC 模式中，M 为 Model，即数据模型，在有网络模块的项目中十分重要，后面的项目中将会了解到 Model 的设计和使用；V 为 View，即视图，用来渲染界面，例如进行组件的风格设置、组件的布局等；C 为 Controller，即控制器，通常我们会把业务逻辑放入 Controller 中。如此分工协作，便很容易将业务逻辑、视图和数据分开，增强项目的封装性与复用性。

首先在 ViewController 文件夹下新建一个命名为 RootViewController.js 的文件，作为整个应用程序的主控制器。其实汇率转换器软件十分简单，只有一个界面，因此这个控制器也是整个项目唯一的控制器。先在其中编写如下代码：

```
import React, { Component } from 'react';
import RootView from './../View/RootView';
export default class RootViewController{
    view(){
        return(
            <RootView />
        );
    }
}
```

上面的代码只给 RootViewController 类实现了一个 view 方法，用来返回应用程序的主界面，后面我们会逐步丰满这个控制器类。

在 View 文件夹下新建一个命名为 RootView.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
  View
} from 'react-native';
export default class RootView extends Component{
    render(){
        return(
            <View style={rootStyle.rootView}>
```

```
                <View style={rootStyle.screenView}></View>
                <View style={rootStyle.keyboardView}></View>
            </View>
        );
    }
}
let rootStyle = StyleSheet.create({
    rootView:{
        flex:1
    },
    screenView:{
        backgroundColor:'rgb(234,86,37)',
        flex:1
    },
    keyboardView:{
        backgroundColor:'rgb(38,41,42)',
        flex:2
    }
});
```

上面的代码简单实现了页面的整体结构，我们将其分为两部分：上半部分占三分之一，用来显示用户的输入信息和汇率的转换信息；下半部分占三分之二，为键盘部分，用来进行用户输入。React Native 有十分强大的布局引擎，因此我们可以十分轻松地实现布局的自适应，无论在 iOS 设备还是 Android 设备，也无论屏幕尺寸如何、设备方向如何，屏幕部分总是会占据整个界面的三分之一，键盘部分总是会占据界面的三分之二。

修改 index.ios.js 与 index.android.js 文件如下：

```
import React, { Component } from 'react';
import {
  AppRegistry,
} from 'react-native';
import RootViewController from './ViewController/RootViewController';
export default class Calu extends Component {
  constructor(props){
    super(props);
    this.rootController = new RootViewController();
  }
  render() {
    return this.rootController.view();
  }
}
AppRegistry.registerComponent('Calu', () => Calu);
```

运行工程，竖屏和横屏模式下的界面效果如图 9-2 与图 9-3 所示。

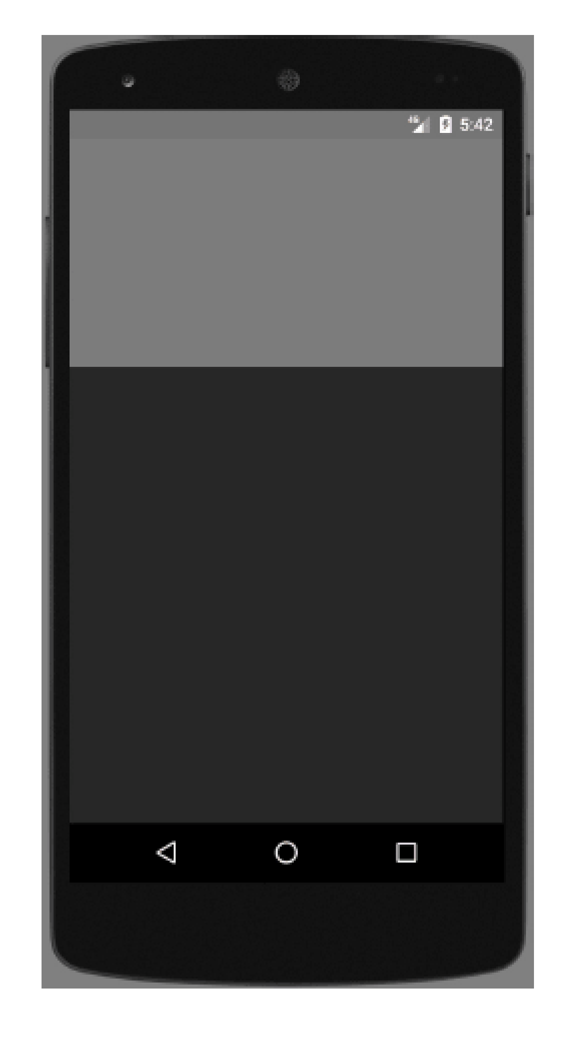


图 9-2　竖屏模式下的主界面布局

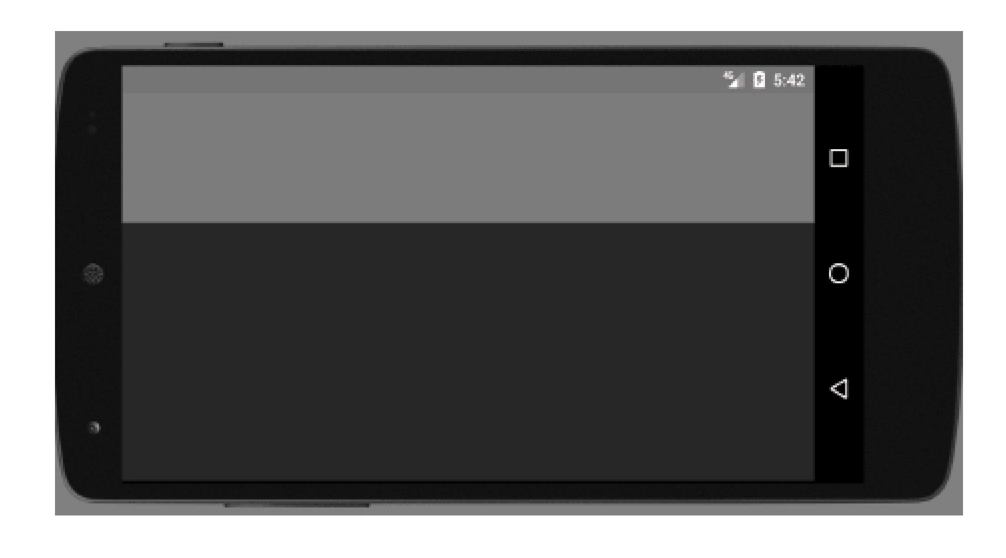


图 9-3　横屏模式下的主界面布局

9.2　显示屏面板的初步开发

本节我们将对汇率计算器的屏幕面板进行初步的界面开发。首先屏幕面板上面有 4 行文字，可分为两类，一类用于描述货币类型，一类用于显示货币数量，除了这 4 行文本外，显示屏的中间还有一条分割线和一个切换按钮，我们可以对分割线和切换按钮进行进一步的封装。

需要注意，在横屏和竖屏模式下，显示屏的具体高度并不一样，因此我们需要监听屏幕的切换来对界面进行重新布局。View 组件的 onLayout 属性就是当界面布局改变时被调用的，我们可以在其中进行显示屏布局的重新处理。

编写 RootView.js 文件如下：

```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
  View,
  Text,
  Dimensions
} from 'react-native';
export default class RootView extends Component{

    constructor(props){
        super(props);
        this.state={
            topText:"从美元",
            bottomText:"到人民币",
            dollar:"659$",
```

```
                RMB:"4415.30￥",
                screenStyle:rootStyle,
            };
        }
        render(){
            return(
                <View style={this.state.screenStyle.rootView}
onLayout={this._onlayout}>
                    <View style={this.state.screenStyle.screenView}>
                        <Text style={this.state.screenStyle.titleText}>
{this.state.topText}</Text>
                        <Text style={this.state.screenStyle.numText}>
{this.state.dollar}</Text>
                        <View style={{height:20}}>
                        </View>
                        <Text style={this.state.screenStyle.titleText}>
{this.state.bottomText}</Text>
                        <Text style={this.state.screenStyle.numText}>
{this.state.RMB}</Text>
                    </View>
                    <View style={this.state.screenStyle.keyboardView}></View>
                </View>
            );
        }
        _onlayout=()=>{
            let {width,height} = Dimensions.get('window');
            if (width>height) {
                this.setState({
                    screenStyle:rootStyle2,
                });
            }else{
                this.setState({
                    screenStyle:rootStyle,
                });
            }
        }

    }

    var rootStyle = StyleSheet.create({
        rootView:{
            flex:1
        },
        screenView:{
            backgroundColor:'rgb(234,86,37)',
```

```
            flex:1,
            paddingTop:22
        },
        keyboardView:{
            backgroundColor:'rgb(38,41,42)',
            flex:2
        },
        titleText:{
            textAlign:'right',
            fontSize:20,
            color:'white',
            marginRight:20
        },
        numText:{
            textAlign:'right',
            fontSize:27,
            color:'white',
            marginRight:20,
            marginTop:10
        }
    });
    let rootStyle2 = StyleSheet.create({
        rootView:{
            flex:1
        },
        screenView:{
            backgroundColor:'rgb(234,86,37)',
            flex:1,
            paddingTop:10
        },
        keyboardView:{
            backgroundColor:'rgb(38,41,42)',
            flex:2
        },
        titleText:{
            textAlign:'right',
            fontSize:14,
            color:'white',
            marginRight:20
        },
        numText:{
            textAlign:'right',
            fontSize:22,
            color:'white',
            marginRight:20,
```

```
            marginTop:2
        }
    });
```

上面的代码在判断设备横竖屏时使用到一个小技巧，即当设备宽度大于高度时认为是横屏，当设备高度大于宽度时认为是竖屏。运行工程，横竖屏效果如图 9-4 与图 9-5 所示。

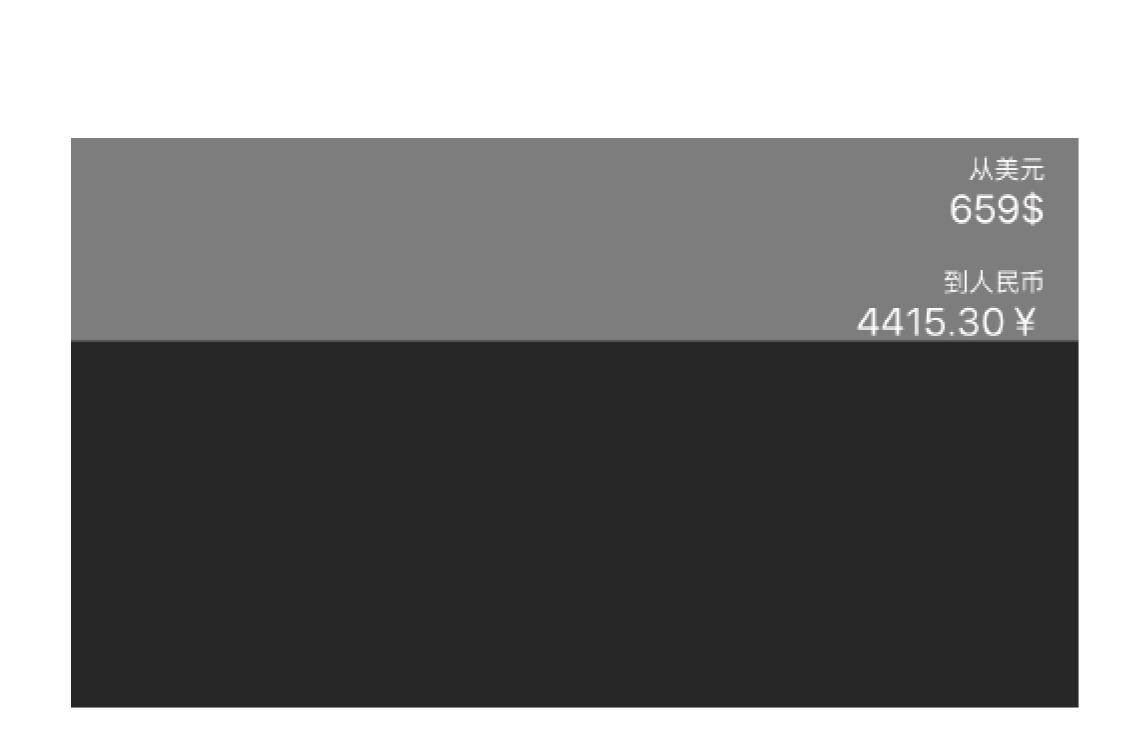


图 9-4　横屏模式



图 9-5　竖屏模式

9.3　货币类型切换功能开发

9.2 节我们完成了显示屏上货币名称与数值文案组件的布局，本节需要完成显示屏的唯一逻辑功能，即切换要转换的货币类型，在做这个功能时，我们首先应该设置布局，接着 9.2 节的代码，在预留的中间 View 中填充转换按钮和分割线组件，回忆下我们所学习的布局技术，我们可以将这个视图设置为水平布局，左边放置转换按钮，右边放置分割线，实现代码如下：

```
            return(
                <View style={this.state.screenStyle.rootView} onLayout=
{this._onlayout}>
                    <View style={this.state.screenStyle.screenView}>
                        <Text style={this.state.screenStyle.titleText}>
{this.state.topText}</Text>
                        <Text style={this.state.screenStyle.numText}>
{this.state.dollar}</Text>
                        <View style={this.state.screenStyle.changeView}>
                            <TouchableHighlight underlayColor='rgb(234,86,37)'
onPress={()=>{
                                    this.props.controller.change();
```

```
                                }}>
                                <Image style={this.state.screenStyle.touchView}
source={require("../src/exchange.png")} />
                            </TouchableHighlight>
                            <View style={this.state.screenStyle.lineView}>
</View>
                        </View>
                        <Text style={this.state.screenStyle.titleText}>
{this.state.bottomText}</Text>
                        <Text style={this.state.screenStyle.numText}>
{this.state.RMB}</Text>
                    </View>
                    <View style={this.state.screenStyle.keyboardView}></View>
                </View>
            );
```

抽 丝 剥 茧

切记不要忘记引入 Image 组件和 TouchableHighlight 组件。上面还省略了向项目中导入图片素材的步骤，你需要在项目根目录中新建一个命名为 src 的文件夹，将本书提供的关于本项目的素材导入进去。

你一定注意到了，上边的代码在实现按钮触发方法的时候使用到了这样的代码：

```
    this.props.controller.change();
```

这其实就是我们所说的逻辑的分离，我们将转换货币类型的逻辑放入 Controller 中进行处理，这里先提前对 RootViewController 进行使用，后面我们会修改 RootViewController 中的代码。

直接运行工程并不能达到预期的效果，需要添加几个样式表，代码如下：

```
    var rootStyle = StyleSheet.create({
        rootView:{
            flex:1
        },
        screenView:{
            backgroundColor:'rgb(234,86,37)',
            flex:1,
            paddingTop:22
        },
        keyboardView:{
            backgroundColor:'rgb(38,41,42)',
            flex:2
        },
        titleText:{
            textAlign:'right',
            fontSize:20,
            color:'white',
```

```
            marginRight:20
        },
        numText:{
            textAlign:'right',
            fontSize:27,
            color:'white',
            marginRight:20,
            marginTop:10
        },
        changeView:{
            flexDirection:'row',
        },
        touchView:{
            width:18,
            height:18,
            marginTop:1,
            marginLeft:70
        },
        lineView:{
            backgroundColor:'white',
            height:1,
            flex:1,
            marginTop:9,
            marginLeft:20
        }
    });
    let rootStyle2 = StyleSheet.create({
        rootView:{
            flex:1
        },
        screenView:{
            backgroundColor:'rgb(234,86,37)',
            flex:1,
            paddingTop:10
        },
        keyboardView:{
            backgroundColor:'rgb(38,41,42)',
            flex:2
        },
        titleText:{
            textAlign:'right',
            fontSize:14,
            color:'white',
            marginRight:20
        },
```

```
        numText:{
            textAlign:'right',
            fontSize:22,
            color:'white',
            marginRight:20,
            marginTop:2
        },
        changeView:{
            flexDirection:'row',
        },
        touchView:{
            width:18,
            height:18,
            marginTop:1,
            marginLeft:300
        },
        lineView:{
            backgroundColor:'white',
            height:1,
            flex:1,
            marginTop:9,
            marginLeft:20
        }
    });
```

此时运行工程，效果如图 9-6 所示。



图 9-6　添加转换按钮

界面布局完成只是我们本节要做的工作的一半，还需要为其添加上转换功能，即如果当前是从美元到人民币，那么当用户单击转换按钮后，界面应该更新为从人民币到美元，如果当前是从人民币到美元，那么用户单击转换按钮后，界面应该更新为从美元到人民币。首先应该在 RootView 类中添加一个属性来标记当前的模式，添加一个方法来进行界面更新，代码如下：

```
    isDollarToRMB = true;
    exchange(isDollar){
        if (!isDollar){
            this.setState({
                topText:"从人民币",
                bottomText:"到美元",
            });
        }else{
            this.setState({
                topText:"从美元",
                bottomText:"到人民币",
            });
        }
        this.isDollarToRMB = !this.isDollarToRMB;
    }
```

到此 RootView 类的开发先告一段落，我们需要修改 RootViewController，代码如下：

```
export default class RootViewController extends Component{
    render(){
        return(
            <RootView controller={this} ref='rootView'/>
        );
    }
    change=()=>{
        if (this.refs.rootView.isDollarToRMB) {
            this.refs.rootView.exchange(false);
        }else{
            this.refs.rootView.exchange(true);
        }

    }
}
```

让 RootViewController 继承自 Component 组件后，你可以方便地操作 ref 来获取界面中的子组件，当用户单击转换按钮时，TouchableHighlight 的触发函数会调用 RootViewController 实例的 change 方法，在这个方法中进行逻辑判断，并重新刷新 RootView 界面。再次运行工程，单击转换按钮，已经可以看到转换效果。

9.4 键盘界面设计

分析键盘界面，其实际上是由一些按钮排列组合而成，因此我们可以先进行键盘按钮的设计，在 View 文件夹下新建一个命名为 NumButton.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  StyleSheet,
  Text,
  Image,
  TouchableHighlight,
  View
} from 'react-native';
export default class NumberButton extends Component{
    render(){
        if (this.props.model==="text") {
            return(
                <TouchableHighlight style={[this.props.style,
{flexDirection:'row'}]} onPress={()=>{
                }}>
                    <Text style={{textAlign:'center',flex:1,
alignSelf:'center',color:'white',fontSize:22}}>{this.props.title}</Text>
                </TouchableHighlight>
            );
        }else{
            return(
                <TouchableHighlight style={[this.props.style,
{flexDirection:'row'}]}>
                    <View style={{alignSelf:'center',flex:1}}>
                        <Image style={{alignSelf:'center', width:45,
height:30}} source={require('../src/delete.png')}/>
                    </View>
                </TouchableHighlight>
            );
        }


    }
}
```

上面的代码将键盘按钮分为了两类，一类是纯文本的按钮，一类是图标按钮。在设计键盘按钮时，需要注意哪些属性要封装在内部、哪些属性要由调用方提供，例如，键盘中文字或图标的居中对齐相关的风格属性应该封装在 NumberButton 类内部，但是按钮的布局、颜色和显示的文案或图标则应该由调用方来设置，这样的组件才有复用性。

修改 RootView 类的 constructor 方法，在其中进行按钮的循环创建：

```
    constructor(props){
        super(props);
        this.state={
            topText:"从美元",
            bottomText:"到人民币",
```

```
                dollar:"659$",
                RMB:"4415.30￥",
                screenStyle:rootStyle,
            };
            var array = new Array();
            var titles = ["1","2","3","delete","4","5","6","0","7","8","9",".",
"C","-","+","="]
            for (var i = 0; i < 16; i++) {
                let element;
                if (i==3) {
                    element = (<NumButton key={i} style={{flex:1,
alignSelf:'stretch'}} source="../src/delete.png" model="image"
title={titles[i]}/>);
                }else if(i==12||i==15){
                    element = (<NumButton key={i} style={{flex:1,
alignSelf:'stretch',backgroundColor:'rgb(234,86,37)'}} title={titles[i]}
model="text" />);
                }
                else{
                    element = (<NumButton key={i} style={{flex:1,
alignSelf:'stretch'}} title={titles[i]} model="text" />);
                }
                array.push(element);
            }
            this.numberButton = array;
        }
```

键盘上的按钮总共分为 4 行，每行 4 列，在布局时，我们可以将每 4 个按钮包装在一个行 View 组件中，实现 RootView 类的 render 方法如下：

```
        render(){
            return(
                <View style={this.state.screenStyle.rootView} onLayout=
{this._onlayout}>
                    <View style={this.state.screenStyle.screenView}>
                        <Text style={this.state.screenStyle.titleText}>
{this.state.topText}</Text>
                        <Text style={this.state.screenStyle.numText}>
{this.state.dollar}</Text>
                        <View style={this.state.screenStyle.changeView}>
                            <TouchableHighlight underlayColor='rgb(234,86,37)'
onPress={()=>{
                                    this.props.controller.change();
                                }}>
                                <Image style={this.state.screenStyle.
touchView}/>
```

```
                            </TouchableHighlight>
                            <View style={this.state.screenStyle.lineView}>
</View>
                        </View>
                        <Text style={this.state.screenStyle.titleText}>
{this.state.bottomText}</Text>
                        <Text style={this.state.screenStyle.numText}>
{this.state.RMB}</Text>
                    </View>
                    <View style={this.state.screenStyle.keyboardView}>
                        <View style={this.state.screenStyle.rowView}>
                            {
                                this.numberButton.slice(0,4)
                            }
                        </View>
                        <View style={this.state.screenStyle.rowView}>
                            {
                                this.numberButton.slice(4,8)
                            }
                        </View>
                        <View style={this.state.screenStyle.rowView}>
                            {
                                this.numberButton.slice(8,12)
                            }
                        </View>
                        <View style={this.state.screenStyle.rowView}>
                            {
                                this.numberButton.slice(12,16)
                            }
                        </View>
                    </View>
                </View>
            );
        }
```

最后，进行样式的编写，无论横屏竖屏，下列的样式都是通用的：

```
        rowView:{
            flexDirection:'row',
            flex:1
        },
        numButtonStyle:{
            flex:1,
        }
```

运行工程，效果如图 9-7 所示。

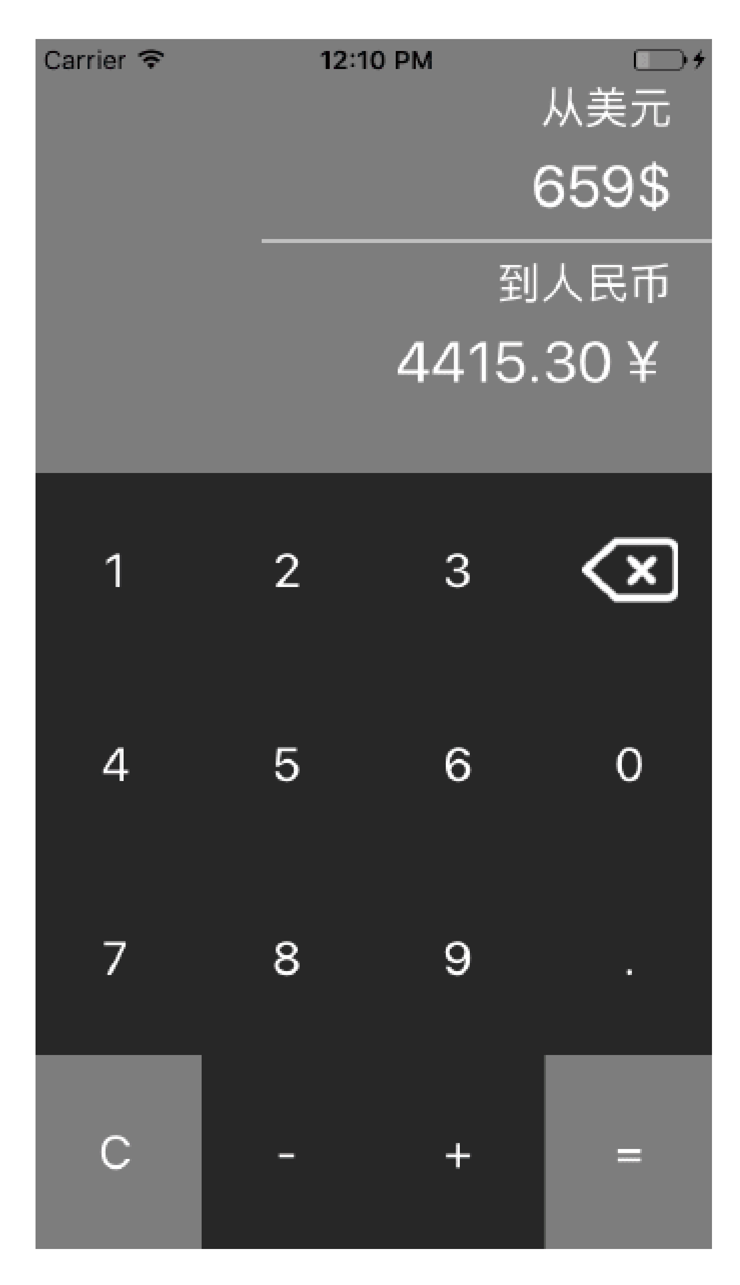


图 9-7　设计键盘界面

9.5　实现汇率转换器核心功能

汇率转换器的核心功能无非是用户输入某种货币的数量，单击等号按钮后，将其转换为另一种货币的数量。在具体编写代码时，我们可以将键盘按钮的功能分类处理。首先修改 NumberButton 类，将按钮的触发逻辑交由上层控制器处理：

```
        render(){
            if (this.props.model==="text") {
                return(
                    <TouchableHighlight style={[this.props.style,
flexDirection:'row'}]} onPress={()=>{
                        this.props.controller.click(this.props.title);
                    }}>
                        <Text style={{textAlign:'center',flex:1, lignSelf:
'center',color:'white',fontSize:22}}>{this.props.title}</Text>
                    </TouchableHighlight>
                );
            }else{
                return(
                    <TouchableHighlight style={[this.props.style,
flexDirection:'row'}]} onPress={()=>{
                        this.props.controller.click(this.props.title);
                    }}>
```

```
                        <View style={{alignSelf:'center',flex:1}}>
                            <Image style={{alignSelf:'center', idth:45,
height:30}} source={require('../src/delete.png')}/>
                        </View>
                    </TouchableHighlight>
                );
            }
        }
```

在 RootView 类中创建键盘按钮时，同时绑定控制器，代码如下：

```
    if (i==3) {
        element = (<NumButton controller={this.props.controller} key={i}
style={{flex:1,alignSelf:'stretch'}} title={titles[i]}
source="../src/delete.png" model="image" />);
    }else if(i==12||i==15){
        element = (<NumButton controller={this.props.controller} key={i}
style={{flex:1,alignSelf:'stretch',backgroundColor:'rgb(234,86,37)'}}
title={titles[i]} model="text" />);
    }else{
        element = (<NumButton controller={this.props.controller} key={i}
style={{flex:1,alignSelf:'stretch'}} title={titles[i]} model="text" />);
    }
```

在 RootViewController 中实现 click 方法：

```
        click=(title)=>{
            if (title==="0"||title==="1"||title==="2"||title==="3"||
title==="4"||
                title==="5"||title==="6"||title==="7"||title==="8"||title==="9") {
                this.refs.rootView.inputNum(title);
            }else if(title==='.'){
                this.refs.rootView.inputDot();
            }else if(title==='delete'){
                this.refs.rootView.delete();
            }else if(title==="C"){
                this.refs.rootView.clear();
            }else if(title==="+"){
                this.refs.rootView.add();
            }else if(title==="-"){
                this.refs.rootView.sub();
            }else if(title==="="){
                this.refs.rootView.cal();
            }
        }
```

在 RootView 类中实现相关方法：

```
    inputNum(num){
        let str = this.state.dollar.slice(0,-1);
        if (this.isDollarToRMB) {
            this.setState({
                dollar:Number(str+num)+'$'
            });
        }else{
            this.setState({
                dollar:Number(str+num)+'￥'
            });
        }
    }
    clear(){
        if (this.isDollarToRMB) {
            this.setState({
                dollar:'0$',
                RMB:'0￥'
            });
        }else{
            this.setState({
                dollar:'0￥',
                RMB:'0$'
            });
        }
    }
    delete(){
        let str = this.state.dollar.slice(0,-1);
        if (str.length===1) {
            if (this.isDollarToRMB) {
                this.setState({
                    dollar:'0$'
                });
            }else{
                this.setState({
                    dollar:'0￥'
                });
            }
        }else{
            if (this.isDollarToRMB) {
                this.setState({
                    dollar:str.substring(0,str.length-1)+'$'
                });
            }else{
                this.setState({
                    dollar:str.substring(0,str.length-1)+'￥'
```

```
                });
            }
        }
    }
    add(){
        let str = this.state.dollar.slice(0,-1);
        if (this.isDollarToRMB) {
            this.setState({
                dollar:Number(str)+1+'$'
            });
        }else{
            this.setState({
                dollar:Number(str)+1+'¥'
            });
        }
    }
    cal(){
        let str = this.state.dollar.slice(0,-1);
        if (this.isDollarToRMB) {
            this.setState({
                RMB:(Number(str)*6.7).toFixed(2)+"¥"
            });
        }else{
            this.setState({
                RMB:(Number(str)/6.7).toFixed(2)+"¥"
            });
        }
    }
    sub(){
        let str = this.state.dollar.slice(0,-1);
        if (str==="0") {
            return;
        }
        if (this.isDollarToRMB) {
            this.setState({
                dollar:Number(str)-1+'$'
            });
        }else{
            this.setState({
                dollar:Number(str)-1+'¥'
            });
        }
    }
    inputDot(){
        let str = this.state.dollar.slice(0,-1);
```

```
            if (str.indexOf(".")!==-1) {
                return;
            }
            if (this.isDollarToRMB) {
                this.setState({
                    dollar:str+'.$'
                });
            }else{
                this.setState({
                    dollar:str+'.¥'
                });
            }
        }
```

上面的代码都是基础的 JavaScript 逻辑实现，理解起来并没有太大的难度。至此，第一款完整的 React Native 应用就开发完成了，把它展示给你的伙伴们吧！

第 10 章

实战项目：微信热门精选

本章我们开始编写一款 React Native 网络应用。微信是现在生活中一个很火的社交工具，其流行原因除了用户可以方便地进行社交行为外，还有强大的公众号和原创文章可以为用户提供各种各样的热门资讯。本章我们就调用微信热门文章的接口服务来开发一款资讯文章推荐阅读器。

10.1 申请免费的 API 服务

在实际工作中，API 服务一般会有专门的开发人员负责，在学习阶段，我们可以借用互联网上的免费 API 服务来搭建客户端项目。天行数据网站提供了许多 API 服务，其并不是完全免费的，对于新注册的用户，可以免费试用 10000 次的请求，对于学习来说，这已经足够了。

访问 https://www.tianapi.com/网址来到天行数据网站的首页，如图 10-1 所示。



图 10-1　天行数据网站首页

若要使用网站提供的 API 服务，则需要申请成为网站的会员，首先单击首页右上角的“个人中心”按钮，此时会弹出登录注册界面，如图 10-2 所示。

图 10-2　登录注册界面

如果以前没有注册过会员，就需要在注册界面填写相应信息来进行注册，注册完成后登录即可。

登录成功后，你可能会看到如图 10-3 所示的界面，这是因为新注册的账号并没有完成邮箱验证，只有完成邮箱验证后才能使用 API 服务，单击“请点此发送验证链接”选项，之后进入邮箱进行验证即可。（因为需要进行邮箱验证，所示请确保注册时提供的邮箱地址正确。）



图 10-3　为邮箱验证的账号信息

抽 丝 剥 茧

基于互联网的 API 服务都不能保证长期稳定有效，如果你在学习本书时发现示例的 API 已经无法使用也不必着急，按照书中提供的思路再寻找其他可用服务即可。

在天行数据网站上找到热门精选 API 栏目，其中有请求相关参数的定义，如图 10-4 所示。


热门精选API

请求方法：HTTP/HTTPS GET　JSON返回示例　代码参考　错误返回码
数据来源：微信公众平台
接口地址：http://api.tianapi.com/wxnew/?key=APIKEY&num=10
使用帮助：默认返回10条数据，非必填参数请按需传递。
更新周期：1小时/次（晚间00-06不更新）

请求参数	类型	必填	参数位置	描述	备注说明
key	string	是	urlParam	API密钥（请在个人中心获取）	用户自己的key
num	int	是	urlParam	指定返回数量，最大50	10
src	string	否	urlParam	指定来源为某微信公众号	例如：人民日报
rand	int	否	urlParam	参数值为1则随机获取	0
word	string	否	urlParam	检索关键词	上海
page	int	否	urlParam	翻页，每页输出数量跟随num参数	1，若指定文章来源则必带此参数

图 10-4　请求参数定义

其中，key 是最重要的参数，是个人中心界面中分配给你账号的 api key 值；num 参数设置返回的数据条数；rand 参数设置是否是随机的；page 参数用来进行分页。

10.2　搭建项目网络模块

使用终端在指定工作目录中新建一个 React Native 工程，使用如下命令：

```
react-native init WXHot
```

稍等片刻，工程建立完成后，在 iOS 和 Android 平台运行，成功进入欢迎界面，工程初始化完成。在工程中新建一个命名为 net 的文件夹，在其中创建一个命名为 NetTool.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
export default class NetTool {
    //获取文章数据
    getArticle=function(page,callback){
        let url = "http://api.tianapi.com/wxnew/?key=
ef7f04344615b7ff44a8b3aa78aa27f3&num=10&page="+page;
        fetch(url,{
            method:'GET',
        }).then((response)=>{
            return response.json();
        }).then((data)=>{
```

```
            callback(data);
        });
    }
}
```

修改 index.ios.js 文件如下，在其中进行请求的测试：

```
import {
  AppRegistry,
  StyleSheet,
  Text,
  View
} from 'react-native';
import NetTool from './net/NetTool';
export default class WXHot extends Component {
  constructor(props){
    super(props);
    this.netTool = new NetTool();
  }
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}
        onPress={()=>{
          this.netTool.getArticle(0,(data)=>{
              console.log(data);
            })
          }
        }>
          Welcome to React Native!
        </Text>
        <Text style={styles.instructions}>
          To get started, edit index.ios.js
        </Text>
        <Text style={styles.instructions}>
          Press Cmd+R to reload,{'\n'}
          Cmd+D or shake for dev menu
        </Text>
      </View>
    );
  }
}
```

注意，在 iOS 平台要访问 HTTP 接口，需要修改 Info.plist 文件，在其中添加如图 10-5 所示的键值。

Key	Type	Value
▼ Information Property List	Dictionary	(17 items)
Localization native development r...	String	en
Bundle display name	String	WXHot
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	org.reactjs.native.example.$(PRODUCT_NAME:rfc1034identifier)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)
View controller-based status bar a...	Boolean	NO
Privacy - Location When In Use U...	String	
▼ App Transport Security Settings	Dictionary	(2 items)
Allow Arbitrary Loads	Boolean	YES
▶ Exception Domains	Dictionary	(1 item)

图 10-5　添加支持 HTTP 请求的字段

在 iOS 平台运行工程，打开调试模式，单击“Welcome to React Native”，可以看到调试区打印出请求到的数据对象，如图 10-6 所示。

```
▼ Object {code: 200, msg: "success", newslist: Array(10)}
    code: 200
    msg: "success"
  ▼ newslist: Array(10)
    ▼ 0: Object
        ctime: "2017-07-31"
        description: "姚之彼方"
        picUrl: "https://zxpic.gtimg.com/infonew/0/wechat_pics_-32792770.jpg/640"
        title: "古装剧里的毁容级画像！朕的美人怎么成了这样？"
        url: "https://mp.weixin.qq.com/s?src=16&ver=276&timestamp=1501473630&signature=5sKzumDrBdXdbqH0aShPfuNIXPxsoLD6i5MXukmw2bB9SAk8IH2v5RgdhysEJLtb
      ▶ __proto__: Object
    ▶ 1: Object
    ▶ 2: Object
    ▶ 3: Object
    ▶ 4: Object
    ▶ 5: Object
    ▶ 6: Object
    ▶ 7: Object
    ▶ 8: Object
    ▶ 9: Object
      length: 10
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
```

图 10-6　请求结果展示

10.3　搭建文章列表界面

在 10.2 节，我们完成了开发前的准备工作，调试好了 API 服务，本节我们来搭建一个简单的文章列表界面。

首先在工程根目录中新建一个命名为 view 的文件夹，其中用来存放视图相关的文件，在其中新建一个命名为 ArticleView.js 的文件，关于文章列表，我们可以采用 FlatList 组件来搭建，实现 ArticleView.js 文件：

```
import React, { Component } from 'react';
import {
  FlatList,
  StyleSheet,
  View,
  Text
} from 'react-native';
export default class ArticleView extends Component{
    constructor(props){
        super(props);
    }
    render(){
        let key = 0;
        this.props.data.forEach(function(item){
            item.key = key++;
        });
        return(
            <FlatList
                data={this.props.data}
                renderItem={({item})=>{
                    return(<Text>{item.title}</Text>);
                }}
                ItemSeparatorComponent={()=>{
                    return(<View style={articleViewStyle.separatorLine}>
</View>);
                }}
            />
        );
    }
}
let articleViewStyle = StyleSheet.create({
    separatorLine:{
        height:1,
        backgroundColor:'gray',
        marginLeft:15
    }
});
```

注意，FlatList 在加载的时候需要每一个数据源对象中都包含一个 key 属性，由于接口 API 返回的数据中并没有属性 key，因此我们需要在每次获取数据刷新后将所有数据都添加一个 key 属性。

修改 index.ios.js 与 index.android.js 文件如下：

```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
```

```
  Text,
  View
} from 'react-native';
import NetTool from './net/NetTool';
import ArticleView from './view/ArticleView';
export default class WXHot extends Component {
  constructor(props){
    super(props);
    this.netTool = new NetTool();
    this.state={
      data:new Array()
    };
    this.page = 1;
    this.getArticle();
  }
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.navigation}></View>
        <ArticleView data={this.state.data}/>
      </View>
    );
  }
  getArticle(){
    this.netTool.getArticle(this.page,(data)=>{
        this.setState({
          data:data.newslist,
        });
        this.page++;
    })
  }
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  navigation:{
    height:64,
    backgroundColor:'white'
  }
});
AppRegistry.registerComponent('WXHot', () => WXHot);
```

上面的代码中，我们先留下一个 64 个单位高度的导航栏，后面章节我们会再回来单独开发导

航栏。在组件的构造方法中，我们进行了一次网络数据的获取，每次数据获取后，都需要将标记页码的属性 page 进行自增，运行工程，效果如图 10-7 所示。

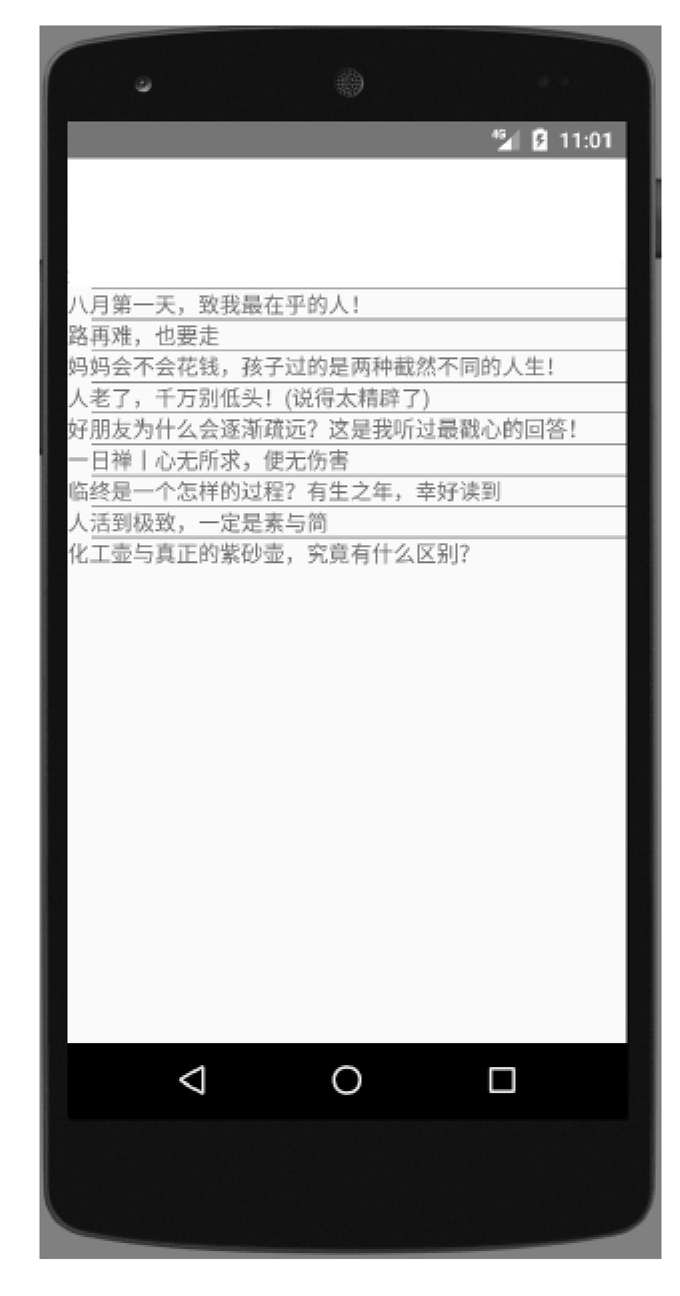


图 10-7　文章列表界面搭建

10.4　文章目录视图与首页导航栏完善

在工程的 view 文件夹下新建一个命名为 ArticleItem.js 的文件，将其作为展现文章目录的视图组件。当用户单击某个列表中的文章时，应该可以跳转到文章详情页，因此使用 TouchableOpacity 组件十分合适。本节将完成界面部分的完善，交互逻辑在后面开发。

在 ArticleItem.js 文件中编写如下代码：

```
import React, { Component } from 'react';
import {
  StyleSheet,
  View,
  Text,
  Image,
  TouchableOpacity
} from 'react-native';
export default class ArticleItem extends Component{
    render(){
        return(
            <TouchableOpacity>
                <View>
```

```
                        <Text style={itemStyle.titleStyle}>
{this.props.data.description}</Text>
                        <View style={itemStyle.contentView}>
                            <Image source={{uri:this.props.data.picUrl}}
style={itemStyle.imageStyle}/>
                            <Text style={itemStyle.contentText}>
{this.props.data.title}</Text>
                            <Text></Text>
                        </View>

                    </View>
                </TouchableOpacity>
            );
        }
    }
    let itemStyle=StyleSheet.create({
        imageStyle:{
            width:80,
            height:120,
            marginLeft:15,
            marginTop:10,
            marginBottom:10
        },
        titleStyle:{
            marginTop:15,
            marginLeft:15,
            fontSize:17
        },
        contentView:{
            flex:1,
            flexDirection:'row'
        },
        contentText:{
            flex:1,
            fontSize:15,
            marginTop:15,
            marginLeft:10,
            marginRight:15
        }
    });
```

有关文章列表所需要显示的数据，我们采用自定义属性的方式来获取，修改 ArticleView 文件如下：

```
    import React, { Component } from 'react';
    import {
```

```
        FlatList,
        StyleSheet,
        View,
        Text
    } from 'react-native';
    import ArticleItem from './ArticleItem';
    export default class ArticleView extends Component{
        constructor(props){
            super(props);
        }
        render(){
            let key = 0;
            this.props.data.forEach(function(item){
                item.key = key++;
            });
            return(
                <FlatList
                    data={this.props.data}
                    renderItem={({item})=>{
                        return(<ArticleItem data={item} />);
                    }}
                    ItemSeparatorComponent={()=>{
                        return(<View style={articleViewStyle.separatorLine}>
</View>);
                    }}
                />
            );
        }
    }
    let articleViewStyle = StyleSheet.create({
        separatorLine:{
            height:1,
            backgroundColor:'gray',
            marginLeft:15
        }
    });
```

将 index.ios.js 与 index.android.js 文件中的 render 方法修改如下，进行导航栏的完善：

```
      render() {
        return (
          <View style={styles.container}>
            <View style={styles.navigation}>
              <Text style={styles.titleStyle}>微信热门推荐</Text>
            </View>
            <ArticleView data={this.state.data}/>
```

```
      </View>
    );
  }
```

新添加的样式表如下：

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  navigation:{
    height:64,
    backgroundColor:'purple'
  },
  titleStyle:{
    color:'white',
    fontSize:22,
    justifyContent:'center',
    alignSelf:'center',
    lineHeight:60
  }
});
```

运行工程，效果如图 10-8 所示，基本的主页界面我们已经搭建完成。



图 10-8　应用主页效果

10.5 文章详情页面的开发

文章详情页面其实十分简单，它就是一个 WebView 组件，用来显示一个网页，本节我们的重点是使用导航进行页面的跳转。注意，在 React Native 0.44 版本后，Navigator 组件被放至单独的模块中，如果你使用的 React Native 版本高于 0.44，就需要手动来安装这个模块，使用 yarn 工具在项目根目录下执行如下命令：

```
yarn add react-native-deprecated-custom-components
```

如果你还没有安装 yarn 工具，可以使用如下命令安装：

```
npm install -g yarn react-native-cli
```

安装完 yarn 工具后，建议设置一下国内镜像，执行命令如下：

```
yarn config set registry https://registry.npm.taobao.org --global
yarn config set disturl https://npm.taobao.org/dist --global
```

准备完成后，在项目的 view 文件夹中新建一个命名为 ArticleDetail.js 的文件，用来展示文章详情页，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  WebView
} from 'react-native';
export default class ArticleDetail extends Component{
    render(){
        let uri =  this.props.uri;
        return(
            <WebView source={{uri:uri}}/>
        );
    }
}
```

上面的代码十分简单，使用 WebView 组件来加载一个网页视图。当用户单击某个文章列表中的目录时，应用会跳转到具体的文章详情页，因此，修改 ArticleItem 类的 render 方法如下，使其向外暴露一个用户交互行为的接口：

```
    render(){
        return(
            <TouchableOpacity onPress={this.props.clickItem}>
                <View>
                    <Text style={itemStyle.titleStyle}>
{this.props.data.description}</Text>
                    <View style={itemStyle.contentView}>
```

```
                            <Image source={{uri:this.props.data.picUrl}}
style={itemStyle.imageStyle}/>
                            <Text style={itemStyle.contentText}>
{this.props.data.title}</Text>
                            <Text></Text>
                        </View>
                    </View>
                </TouchableOpacity>
            );
        }
```

修改 ArticleView 类的 render 方法如下，添加设置 clickItem 属性的代码：

```
        render(){
            let key = 0;
            this.props.data.forEach(function(item){
                item.key = key++;
            });
            return(
                <FlatList
                    data={this.props.data}
                    renderItem={({item})=>{
                        return(<ArticleItem data={item} clickItem={()=>{
                                this.props.goDetails(item);
                            }}/>);
                    }}
                    ItemSeparatorComponent={()=>{
                        return(<View style={articleViewStyle.separatorLine}>
</View>);
                    }}
                />
            );
        }
```

在 index.ios.js 与 index.android.js 文件中修改 WXHot 类的 render 方法如下，在其中进行导航页面的控制：

```
      render() {
        return (
          <Navigator initialRoute={{title:'微信热门推荐',index:0}}
          renderScene={(route,navigator)=>{
            if (route.index==0) {
              return(
               <View style={styles.container}>
                 <View style={styles.navigation}>
                   <Text style={styles.titleStyle}>{route.title}</Text>
                 </View>
```

```
                <ArticleView data={this.state.data} goDetails={(item)=>{
                  navigator.push({
                      title:item.description,
                      index:route.index+1,
                      item:item
                    });
                }}/>
              </View>
            );
          }else{
            return(
              <View style={styles.container}>
                <View style={styles.navigation}>
                  <Text style={styles.detailTitle}>{route.title}</Text>
                  <View style={styles.button}>
                    <Button title="返回" onPress={()=>{
                     navigator.pop();
                  }}/>
                  </View>

                </View>
                <ArticleDetail uri={route.item.url}/>
              </View>
            );
          }
        }}/>
      );
    }
```

使用 Navigator 实例的 push 与 pop 方法可以十分容易地进行场景的切换，相关样式表代码如下：

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  navigation:{
    height:64,
    backgroundColor:'purple',
  },
  titleStyle:{
    color:'white',
    fontSize:22,
    justifyContent:'center',
    alignSelf:'center',
    lineHeight:60
```

```
    },
    detailTitle:{
      position:'absolute',
      color:'white',
      fontSize:17,
      alignSelf:'center',
      lineHeight:60,
      textAlign:'center'
    },
    button:{
      position:'relative',
      width:60,
      height:30,
      marginTop:15,
      marginLeft:15,
    }
  });
```

运行工程，微信热门推荐应用就基本完成了，详情页效果如图 10-9 所示，当然此项目还并不完善，还有下拉刷新和上拉加载更多等功能需要我们开发，后面我们会继续进行完善。



图 10-9　文章详情页

10.6　为文章列表页添加下拉刷新与上拉加载更多功能

大部分网络应用中的列表页都有下拉刷新和上拉加载更多的功能，尤其是一些实时资讯和文章类的应用，就像我们所开发的这款微信热门文章应用。FlatList 组件自带下拉刷新与上拉加载的功能，我们只需要设置相关属性即可。在 FlatList 中，刷新组件的显示与否是由 refreshing 属性控制的，其是一个受控的属性，因此我们需要使用状态来控制它，修改 ArticleView 类的构造方法如下：

```
        constructor(props){
            super(props);
            this.state={
                isRefresh:false
            }
        }
```

修改 render 方法如下：

```
        render(){
            let key = 0;
            this.props.data.forEach(function(item){
                item.key = key++;
            });
            return(
                <FlatList
                    data={this.props.data}
                    renderItem={({item})=>{
                        return(<ArticleItem data={item} clickItem={()=>{
                                this.props.goDetails(item);
                            }}/>);
                    }}
                    ItemSeparatorComponent={()=>{
                        return(<View style={articleViewStyle.separatorLine}>
</View>);
                    }}
                    onRefresh={()=>{
                        if (!this.state.isRefresh) {
                        this.setState({
                            isRefresh:true
                        });
                        this.props.refresh(()=>{
                            this.setState({
                                isRefresh:false
                            });
                        });
                        }
                    }}
                    refreshing={this.state.isRefresh}
                    onEndReached={()=>{
                        this.props.loadMore();
                    }}
                />
            );
        }
```

FlatList 组件的 onEndReached 属性用来设置当列表滑动到接近底部时调用的方法，我们刚好可以使用这个属性来做上拉加载更多的需求。

修改 index.ios.js 文件与 index.android.js 文件中的 ArticleView 组件如下：

```
<ArticleView data={this.state.data} goDetails={(item)=>{
   navigator.push({
       title:item.description,
       index:route.index+1,
       item:item
     });
}}
refresh={(callback)=>{
   this.page=1;
   this.call = callback;
   this.getArticle();
}}
loadMore={()=>{
   this.getArticle();
}}/>
```

前面我们并没有考虑到多页加载的情况，实际上，每次加载新的一页后，都应该将获取到的数据拼接到原有数据之后，除此之外，在刷新数据时，应该将多余的数据清空，修改 getArticle 方法如下：

```
getArticle(){
  this.netTool.getArticle(this.page,(data)=>{
      let oldData = new Array();
      if (this.page==1) {

      }else{
        oldData = this.state.data;
      }
      this.setState({
        data:oldData.concat(data.newslist)
      });
      this.page++;
      if (this.call) {
        this.call();
      }
  })
}
```

运行工程，可以看到下拉刷新效果与上拉加载效果。到此，一个较为完善的 React Native 网络应用就开发完成了，好好享受它吧！

第 11 章

实战项目：掌上新闻

通过第 9 章和第 10 章的学习，相信你对 React Native 应用开发的理解一定更深了一步，前两章的实战项目略微偏简单，从本章开始，我们将一起完成一款完善的 React Native 商业应用：掌上新闻。

本章我们依然要采用天行数据的 API 服务来获取新闻资讯内容，你在上一章申请的天行账号依然可以继续使用。掌上新闻项目分为多个板块，每个板块用来显示一类新闻，除了新闻的展示外，我们还需要添加一些更易用的功能，例如用户感兴趣新闻的收藏。

11.1 应用结构搭建

开始开发一款新的应用程序，尤其是 React Native 应用程序，首先要做的便是界面框架的搭建。使用 react-native init News 命令新建一个 React Native 工程。在其根目录下新建一个命名为 View 的文件夹，用来存放自定义的视图文件。在 View 文件夹中新建 4 个 JavaScript 文件，分别命名为 MainView.js、NavigationBar.js、PageView.js、TitleBar.js。其中，MainView 为应用程序的主界面，NaviagationBar 为可复用的导航栏，PageView 是核心的新闻展示页，TitleBar 为分类标题条。

导航栏作为每个界面顶部的视图元素，往往用来展示页面标题，用来布局一些功能性的按钮。在 NavigationBar.js 文件中简单编写如下代码：

```
import React, { Component } from 'react';
import {
  View,
  StyleSheet
} from 'react-native';
export default class NavigationBar extends Component{
    render(){
```

```
        return(
            <View style={naviStyles.bar}></View>
        );
    }
}
let naviStyles = StyleSheet.create({
    bar:{
        height:64,
        backgroundColor:'red'
    }
});
```

现在我们只需编写一个框架，后面会专门对导航栏进行优化与完善。

在 PageView.js 文件中编写如下代码：

```
import React, { Component } from 'react';
import {
  View,
  StyleSheet,
  ScrollView,
  Dimensions
} from 'react-native';
export default class PageView extends Component{
    render(){
        var {height, width} = Dimensions.get('window');
        return(
            <ScrollView style={mainStyle.pageView} horizontal={true}>
                <View style={{backgroundColor:'red',width:width}}></View>
                <View style={{backgroundColor:'blue',width:width}}></View>
            </ScrollView>
        );
    }
}
let mainStyle = StyleSheet.create({
    pageView:{
        backgroundColor:'white'
    }
});
```

这里我们使用到了前边学习过的 ScrollView 组件，用来展示新闻分类组合页，用户可以通过滑动手势来进行新闻分类的切换。

在 TitleBar.js 文件中编写如下代码：

```
import React, { Component } from 'react';
import {
  View,
```

```
  StyleSheet,
  ScrollView,
  Text
} from 'react-native';
export default class TitleBar extends Component{
    render(){
        return(
            <ScrollView style={titleBarStyles.bar} horizontal={true}>
                <Text style={titleBarStyles.text}>时政要闻</Text>
                <Text style={titleBarStyles.text}>体育新闻</Text>
                <Text style={titleBarStyles.text}>娱乐新闻</Text>
                <Text style={titleBarStyles.text}>时政要闻</Text>
                <Text style={titleBarStyles.text}>体育新闻</Text>
                <Text style={titleBarStyles.text}>娱乐新闻</Text>
                <Text style={titleBarStyles.text}>时政要闻</Text>
                <Text style={titleBarStyles.text}>体育新闻</Text>
                <Text style={titleBarStyles.text}>娱乐新闻</Text>
            </ScrollView>
        );
    }
}
let titleBarStyles = StyleSheet.create({
    bar:{
        backgroundColor:'green',
        maxHeight:30
    },
    text:{
        height:30
    }
});
```

实现了各个子组件的框架搭建后，需要在 MainView.js 文件中对它们进行组合：

```
import React, { Component } from 'react';
import {
  View,
  StyleSheet
} from 'react-native';
import NavigationBar from './NavigationBar';
import TitleBar from './TitleBar';
import PageView from './PageView';
export default class MainView extends Component{
    render(){
        return(
            <View style={{flex:1,backgroundColor:'purple'}}>
                <NavigationBar />
```

```
                <TitleBar />
                <PageView />
            </View>
        );
    }
}
```

一个简易的项目框架就搭建完成了，后面我们只需要分步完善各个子组件即可。修改 index.ios.js 文件与 index.android.js 文件中的 render 方法如下：

```
render() {
  return (
    <MainView />
  );
}
```

运行工程，完成的效果如图 11-1 所示。

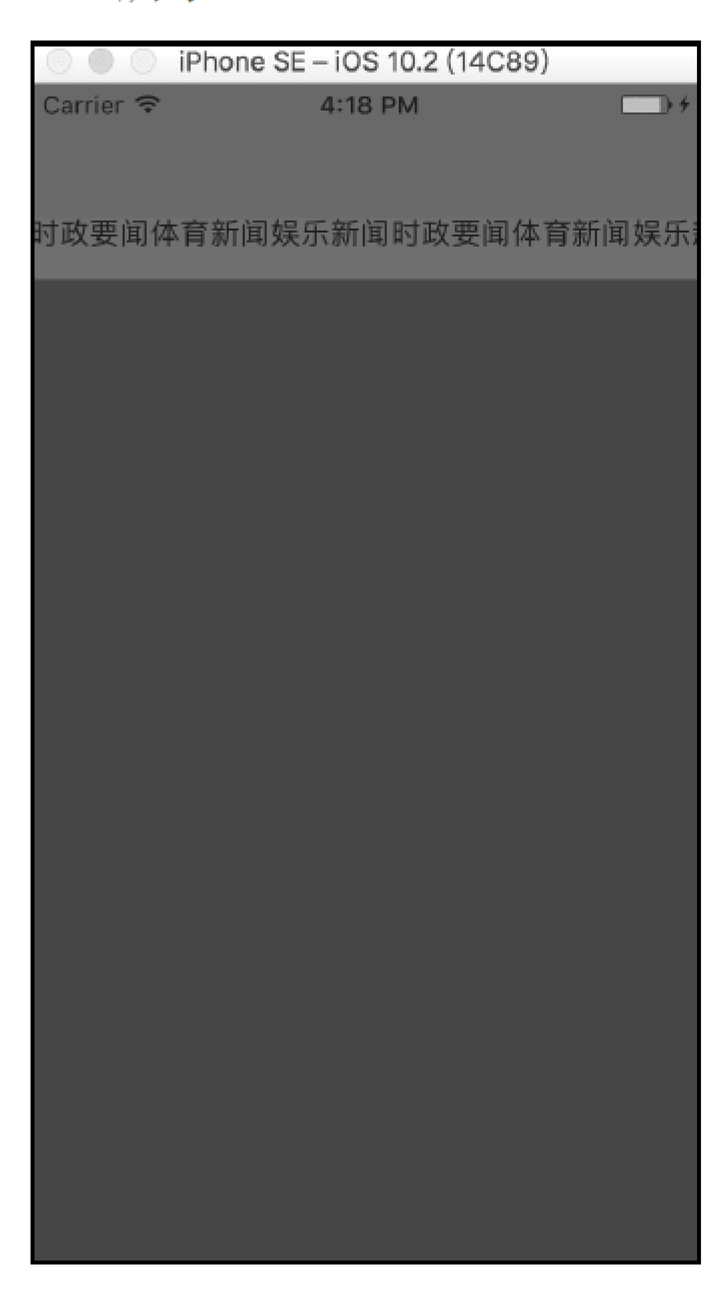


图 11-1　应用界面框架搭建

11.2　完善标题栏组件

我们在 11.1 节搭建了一个简易的应用程序框架，无论是界面还是功能，都还不完善，本节我们将对标题栏组件进行完善。首先，标题栏组件上水平排列一组新闻分类标题，当用户单击某个标题时，相应的新闻会呈现给用户。在设计标题栏时，我们应该考虑这样几个问题：

（1）标题栏上的每个标题都是独立的，可以接收用户事件，并将事件传递出去。

（2）当用户选中某个标题时，此标题应该有 UI 上的差异，可以让用户分辨出当前所选中的栏目。

（3）当用户选中某个标题时，标题栏应该滚动到用户舒适的位置。

除了上面所列举的功能需求外，在编写代码时，我们还需要考虑代码的复用性。实现 TitleBar 类的构造方法如下：

```
constructor(props){
    super(props);
    //标题数组
    this.dataArray = [
        "社会新闻","国内新闻","国际新闻","娱乐要闻","体育新闻",
        "NBA 新闻","足球要闻","科技新闻","创业新闻","苹果新闻",
        "军事新闻","移动互联","旅游资讯","健康知识","奇闻异事",
        "美女图片","VR 科技","IT 资讯"
    ];
    //用来标题用户选中的标题
    let selecteds = new Array();
    for(let i=0;i<this.dataArray.length;i++){
        if (i===0) {
            selecteds.push("rgb(0,0,0)");
        }else{
            selecteds.push("rgb(111,111,111)");
        }
    }
    this.state={
        selected:selecteds,
    };
    this.selectedIndex = 0;
}
```

在 TitleBar 类中实现一个命名为 createTips 的方法，这个方法用来动态创建标题栏上的标题按钮，代码如下：

```
createTips(){
    let tipsArray = new Array();
    for (var i = 0; i < this.dataArray.length; i++) {
        //这里的 index 必须使用 let 声明，var 会产生变量提升
        let index = i;
        let element = (<Text
            onPress={()=>{
                let selecteds = new Array();
                for(let i=0;i<this.dataArray.length;i++){
                    if (i===index) {
                        selecteds.push("rgb(0,0,0)");
```

```
                            }else{
                                selecteds.push("rgb(111,111,111)");
                            }
                        }
                        this.selectedIndex = index;
                        this.setState({
                            selected:selecteds
                        });
                        if (index<this.dataArray.length-4) {
                            this.refs.scrollView.scrollTo({x:75*index,
animated:true});
                        }else{
                            this.refs.scrollView.scrollToEnd({animated: true});
                        }

                    }}
                 key={index} style={[titleBarStyles.text,
{color:this.state.selected[index]}]}>{this.dataArray[i]}</Text>);
                tipsArray.push(element);
            }
            return tipsArray;
        }
```

createTips 方法中有几个点需要读者格外注意，首先 Text 组件 onPress 方法的实现实际上应用了闭包，其中需要使用到当前标题在数组中的下标 index，在声明 index 时，读者务必使用 let 关键字，如果这里使用了 var 关键字，则会产生变量提升，闭包中的 index 将始终等于循环变量 i 最终的值，而不是我们所期望的下标值。

修改 TitleBar 类的 render 方法如下：

```
        render(){
            return(
                <ScrollView style={titleBarStyles.bar} horizontal={true}
                showsHorizontalScrollIndicator={false} ref={"scrollView"}>
                    {this.createTips()}
                </ScrollView>
            );
        }
```

修改样式表如下：

```
    let titleBarStyles = StyleSheet.create({
        bar:{
            maxHeight:30,
            backgroundColor:'rgb(222,222,222)'
        },
        text:{
```

```
            height:30,
            marginLeft:10,
            marginRight:10,
            lineHeight:28,
            color:'rgb(111,111,111)'
        },
    });
```

通过上面的完善，标题栏的功能更加充实，运行工程，效果如图 11-2 所示。

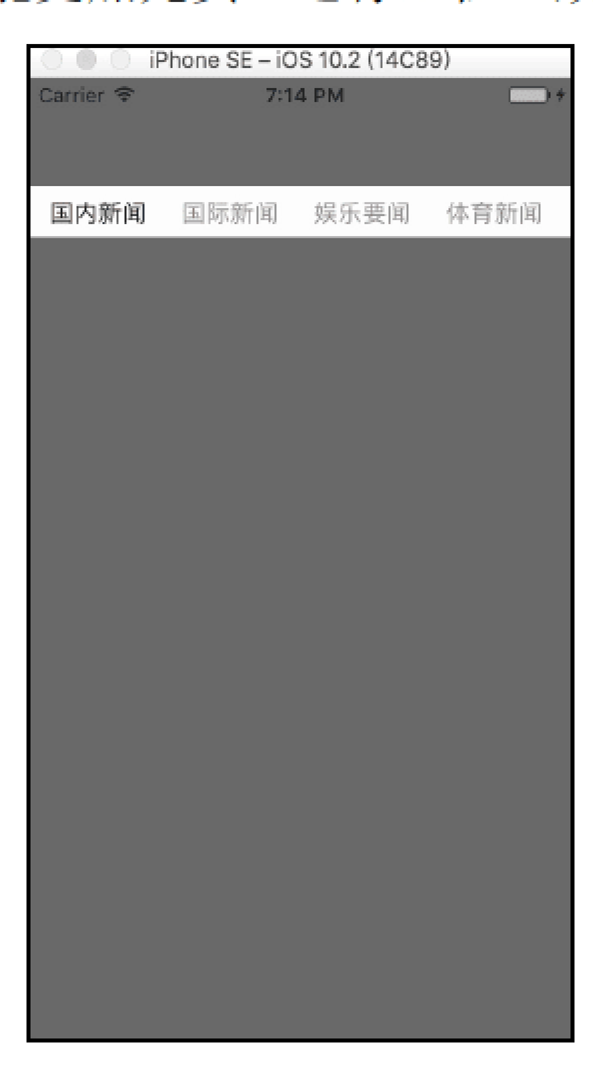


图 11-2　经过优化后的标题栏

11.3　进行网络模块的开发

界面上渲染的新闻资讯数据全部来自网络，因此我们需要封装一个可复用的网络模块，在工程的根目录下新建一个命名为 Net 的文件夹，在其中新建一个命名为 NetTool.js 的文件，编写如下代码：

```
    export default class  NetTool {
        constructor(){
            this.apis = ["https://api.tianapi.com/social/? key=您的 appkey
&num=20&page=",
        "https://api.tianapi.com/guonei/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/world/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/huabian/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/tiyu/? key=您的 appkey &num=20&page=",
       "https://api.tianapi.com/football/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/nba/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/keji/? key=您的 appkey &num=20&page=",
```

```
        "https://api.tianapi.com/startup/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/apple/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/military/?key=您的 appkey&num=20&page=",
        "https://api.tianapi.com/mobile/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/travel/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/health/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/qiwen/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/meinv/? key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/vr/?key=您的 appkey &num=20&page=",
        "https://api.tianapi.com/it/?key=您的 appkey&num=20&page="
            ];
        }
        getNewsData(type,page,callback){
            let url = this.apis[type]+page;
            fetch(url,{
                method:'GET'}).then((response)=>{
                return response.json();
            }).then((data)=>{
                callback(data);
            })
        }
    }
```

由于每一个界面都对应一个网络请求，因此我们可以在构造方法中将请求地址组合成数组。需要注意，其中的参数 key 要换成读者自己在天行数据网站上注册得到的 AppKey 值。通过如下方式调用 getNewsData 方法，可以打开调试模式，观察调试区的打印数据来确认接口的调用是否畅通。

```
            let tool = new NetTool();
            tool.getNewsData(14,1,(data)=>{
                console.log(data);
            });
```

11.4　使用列表展示数据

在 11.3 节中，我们已经可以从互联网上获取需要的数据。将数据展示在对应的分类模块下，可以使用 FlatList 组件。在 PageView.js 文件中导入 FlatList 组件以及用来测试效果的 Text 组件，代码如下：

```
    import {
      View,
      StyleSheet,
      ScrollView,
      Dimensions,
```

```
    FlatList,
    Text
  } from 'react-native';
```

在 PageView 类中实现一个创建列表的方法和请求数据的方法，代码如下：

```
        contentView(width){
            let views = new Array();
            for(let i=0;i<this.netTool.apis.length;i++){
                let element = (<FlatList key={i} style={{width:width}}
data={this.state.dataArray[i]}
                    renderItem={({item})=>{
                        return(<Text>{item.title}</Text>);
                    }}
                />);
                views.push(element);
            }
            return views;
        }
        getData(index,page){
            this.netTool.getNewsData(index,page,(data)=>{
                let list = data.newslist;
                let i = 0;
                list.forEach((item)=>{
                    item.key = i++;
                });
                let array = this.state.dataArray;
                array[index] = list;
                this.setState({
                    dataArray:array
                }
                );
            });
        }
```

修改 PageView 类的 render 方法，代码如下：

```
        render(){
            var {height, width} = Dimensions.get('window');
            return(
                <ScrollView pagingEnabled={true}
                showsHorizontalScrollIndicator={false}
style={mainStyle.pageView} horizontal={true}>
                    {this.contentView(width)}
                </ScrollView>
            );
        }
```

上面的代码中，我们先采用 Text 组件来显示新闻资讯的标题，再完善具体的目录页，运行工程，左右滑动可以看到每个模块都请求到了正确的数据，效果如图 11-3 所示。

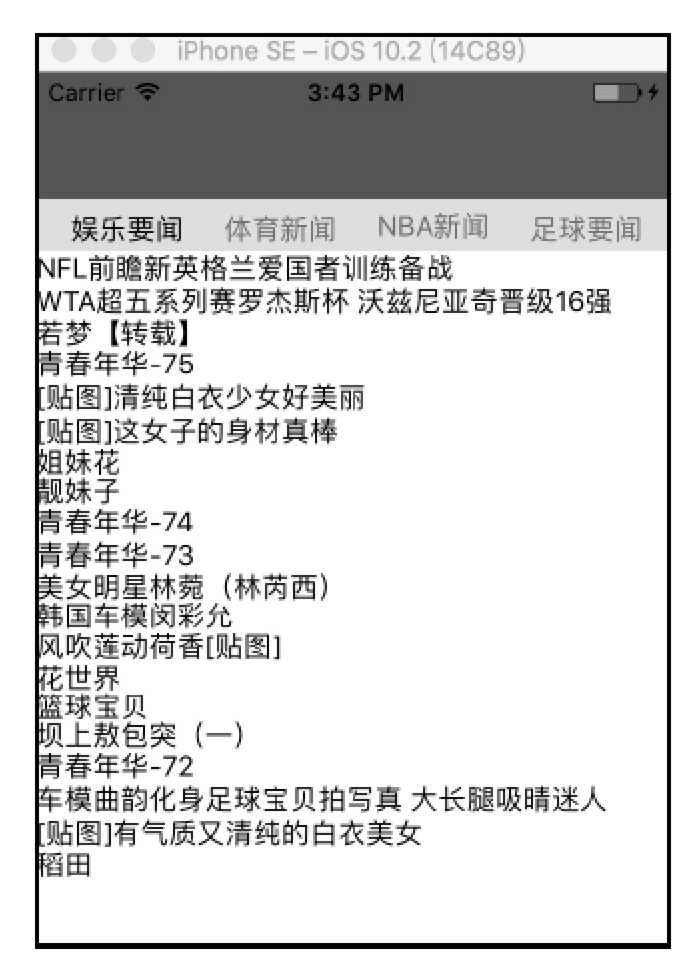
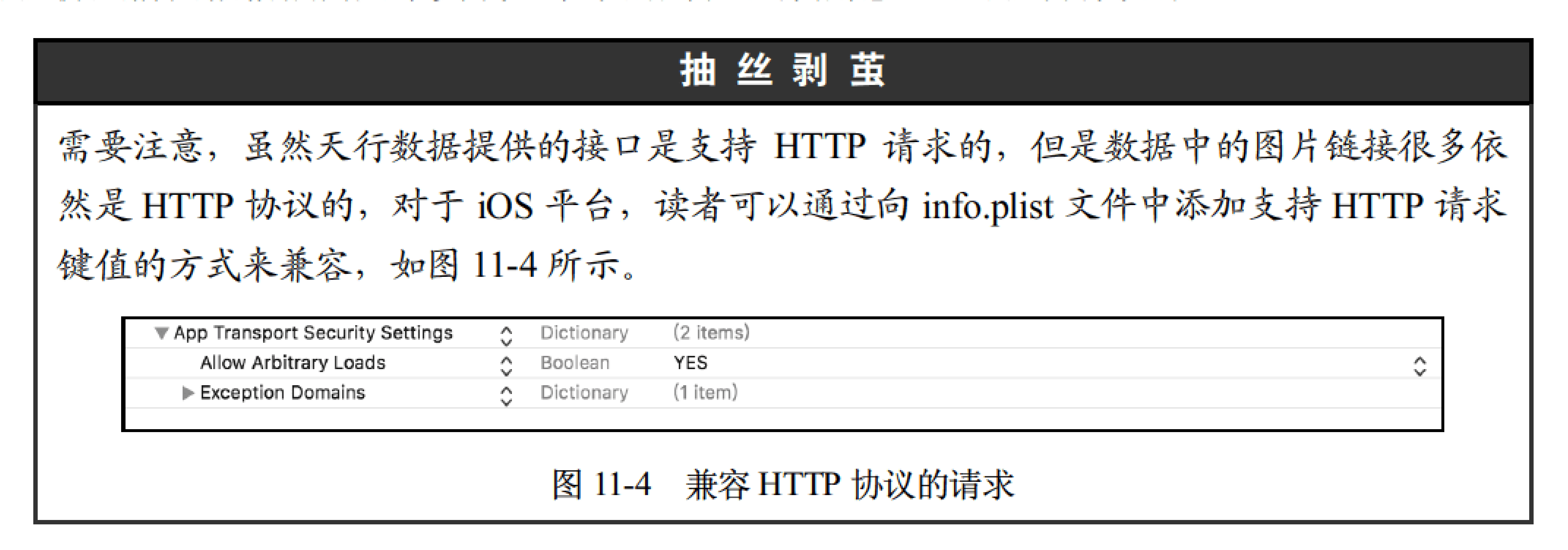


图 11-3　使用列表展示数据

11.5　完善新闻目录列表

完善目录列表对读者来说应该不在话下，天行数据提供的 API 服务的接口结构都类似，读者可以仿照前面微信热门文章实战一章中的列表进行搭建，也可以自行设计。



抽 丝 剥 茧

需要注意，虽然天行数据提供的接口是支持 HTTP 请求的，但是数据中的图片链接很多依然是 HTTP 协议的，对于 iOS 平台，读者可以通过向 info.plist 文件中添加支持 HTTP 请求键值的方式来兼容，如图 11-4 所示。

▼ App Transport Security Settings	Dictionary	(2 items)
Allow Arbitrary Loads	Boolean	YES
▶ Exception Domains	Dictionary	(1 item)

图 11-4　兼容 HTTP 协议的请求

在工程的 View 文件夹下新建一个命名为 ItemView.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {
  View,
  StyleSheet,
  Text,
  Image
} from 'react-native';
export default class ItemView extends Component{
```

```
        render(){
            if (this.props.item.picUrl.length>0) {
            return(
                <View>
                    <Text style={itemStyles.title}>
{this.props.item.description}</Text>
                    <View style={itemStyles.contentView}>
                        <Image style={itemStyles.image} source=
{{uri:this.props.item.picUrl}}/>
                        <Text style={itemStyles.subTitle}>
{this.props.item.title}</Text>
                    </View>
                    <Text style={itemStyles.time}> {this.props.item.ctime}
</Text>
                    <View style={itemStyles.line}></View>
                </View>
            );
            }else{
            return(
                <View>
                   <Text style={itemStyles.title}> {this.props.item.description}
</Text>
                    <Text style={itemStyles.detail}>{this.props.item.title}
</Text>
                    <Text style={itemStyles.time}> {this.props.item.ctime}
</Text>
                    <View style={itemStyles.line}></View>
                </View>
            );
            }
        }
    }
    let itemStyles = StyleSheet.create({
        image:{
            width:80,
            height:120,
            marginLeft:15,
            marginTop:5
        },
        title:{
            fontSize:15,
            marginLeft:15,
            marginTop:10
        },
        contentView:{
```

```
            flexDirection:'row',
        },
        subTitle:{
            marginTop:5,
            marginRight:15,
            marginLeft:5,
            flex:1,
            fontSize:13,
            color:'rgb(111,111,111)'
        },
        detail:{
            flex:1,
            marginTop:10,
            marginLeft:15,
            fontSize:13,
            marginRight:15,
            color:'rgb(111,111,111)'
        },
        time:{
            alignSelf:'flex-end',
            marginRight:15
        },
        line:{
            backgroundColor:'rgb(233,233,233)',
            marginLeft:15,
            height:1,
            marginTop:5
        }
    });
```

最后，不要忘了修改 PageView 类中的 contentView 方法：

```
        contentView(width){
            let views = new Array();
            for(let i=0;i<this.netTool.apis.length;i++){
                let element = (<FlatList key={i} style={{width:width}}
data={this.state.dataArray[i]}
                    renderItem={({item})=>{
                        return(<ItemView item={item} />);
                    }}
                />);
                views.push(element);
            }
            return views;
        }
```

上面的代码根据数据中有无图片提供了两种渲染方式。注意，由于互联网数据具有不稳定性，因此新闻数据并不一定有效，读者如果遇到问题，无须纠结。

运行工程，效果如图 11-5 所示。



图 11-5　新闻列表界面

11.6　标题栏与页面联动开发与优化加载逻辑

到 11.5 节为止，掌上新闻应用程序首页的核心元素都已经开发完成，但是它们之间依然是独立的，也就是说，当用户选中标题栏上某个栏目后，列表界面并没有滚动到相应的分类模块，到用户手动滚动到某个新闻模块时，标题栏上的选中标题也没有同步修改。

首先将 TitleBar 类中的 createTips 方法修改如下，将其中关于标题选择的逻辑抽离出来：

```
        createTips(){
            let tipsArray = new Array();
            for (var i = 0; i < this.dataArray.length; i++) {
                //这里的 index 必须使用 let 声明，var 会产生变量提升
                let index = i;
                let element = (<Text
                    onPress={()=>{
                        this.selected(index);
                        if (index<this.dataArray.length-4) {
                            this.refs.scrollView.scrollTo({x:75*index,
animated:true});
                        }else{
                            this.refs.scrollView.scrollToEnd({animated: true});
```

```
                        }
                        //进行联动
                        this.props.click(index);
                    }}
                key={index} style={[titleBarStyles.text,
{color:this.state.selected[index]}]}>{this.dataArray[i]}</Text>);
                tipsArray.push(element);
            }
            return tipsArray;
        }
```

实现 selected 方法如下：

```
        selected(index){
            let selecteds = new Array();
            for(let i=0;i<this.dataArray.length;i++){
                if (i===index) {
                    selecteds.push("rgb(0,0,0)");
                }else{
                    selecteds.push("rgb(111,111,111)");
                }
            }
            this.selectedIndex = index;
            this.setState({
                selected:selecteds
            });
            if (index<this.dataArray.length-4) {
                this.refs.scrollView.scrollTo({x:75*index,animated:true});
            }else{
                this.refs.scrollView.scrollToEnd({animated: true});
            }
        }
```

在 MainView 类中进行 TitleBar 与 PageView 的逻辑关联：

```
    export default class MainView extends Component{
        render(){
            return(
                <View style={{flex:1}}>
                    <NavigationBar />
                    <TitleBar click={this.clickTitle} ref="titleBar"/>
                    <PageView ref='pageView' scrollEnd={this.scrollEnd}/>
                </View>
            );
        }
        clickTitle=(index)=>{
            this.refs.pageView.show(index);
```

```
        }
        scrollEnd = (index)=>{
            this.refs.titleBar.selected(index);
        }
    }
```

需要特别注意，上面的 clickTitle 函数与 scrollEnd 函数必须以箭头函数的方式创建，这样其中的 this 才能指向 MainView 实例。

在 PageView 类中修改 render 方法并添加 show 方法：

```
        render(){
            var {height, width} = Dimensions.get('window');
            return(
                <ScrollView pagingEnabled={true}
                showsHorizontalScrollIndicator={false} style=
{mainStyle.pageView} horizontal={true} ref={'scrollView'}
                onMomentumScrollEnd={(param)=>{
                    let index = param.nativeEvent.contentOffset.x/width;
                    this.props.scrollEnd(index);
                }}
                >
                    {this.contentView(width)}
                </ScrollView>
            );
        }
        show(index){
            var {height, width} = Dimensions.get('window');
            this.refs.scrollView.scrollTo({x:index*width,animated:true});
        }
```

show 方法很好理解，其根据用户选中的标题栏 index 来操作当前的 scrollView 滚动到合适的位置，ScrollView 组件的 onMomentumScrollEnd 会在滚动动画结束后被调用，从这个回调方法的参数中可以获取到当前 ScrollView 组件滚动到的位置，通过它我们可以进行标题栏的联动。

我们前面在填充界面数据时，一次将所有栏目的数据全部进行了请求和渲染，这样做是十分愚蠢的，首先同时发送大量的请求是很费时间的，其次当请求返回后，同时渲染所有栏目界面又是非常耗性能的，在极端情况下可能会造成界面假死的现象。正确的做法是在应用程序启动时，请求和下载第一个栏目的数据即可，后面当用户浏览到某个栏目时再相应地加载数据。修改 PageView 类的构造方法，去掉循环加载逻辑：

```
        constructor(props){
            super(props);
            this.netTool =  new NetTool();
            let dataArray = new Array();
            for (let i = 0; i < this.netTool.apis.length; i++) {
                dataArray.push([]);
```

```
        }
        this.state={
            dataArray:dataArray
        }
        this.getData(0,1);
    }
```

在 render 方法中加入请求当前页数据的逻辑：

```
    render(){
        var {height, width} = Dimensions.get('window');
        return(
            <ScrollView pagingEnabled={true}
            showsHorizontalScrollIndicator={false}
style={mainStyle.pageView} horizontal={true} ref={'scrollView'}
            onMomentumScrollEnd={(param)=>{
                let index = param.nativeEvent.contentOffset.x/width;
                this.props.scrollEnd(index);
                if (this.state.dataArray[index].length==0) {
                    this.getData(index,1);
                }
            }}
            >
                {this.contentView(width)}
            </ScrollView>
        );
    }
```

注意，onMomentumScrollEnd 回调在 iOS 平台上无论是用户手动滑动 ScrollView 还是开发者通过代码来滑动 ScrollView 都会被触发，但是在 Android 平台上只有用户的滑动行为会触发，因此针对 Android 平台，还需要修改 show 方法：

```
    show(index){
        var {height, width} = Dimensions.get('window');
        this.refs.scrollView.scrollTo({x:index*width,animated:true});
        if (Platform.OS === 'android') {
            if (this.state.dataArray[index].length==0) {
                this.getData(index,1);
            }
        }
    }
```

抽 丝 剥 茧

Platform 是 React Native 中用来分区平台的对象，不要忘记使用前先引入哦。

11.7 使用导航进行页面跳转

新闻资讯的详情页可以直接采用 WebView 来设计，页面的跳转我们依然使用 Navigator 组件来实现，首先需要在工程中安装 Navigator 组件，在工程根目录中执行如下命名：

```
yarn add react-native-deprecated-custom-components
```

将 ItemView 实现成可以接收用户交互的组件，在其中引入 TouchableOpacity 组件：

```
import {
  View,
  StyleSheet,
  Text,
  Image,
  TouchableOpacity
} from 'react-native';
```

修改 render 方法：

```
    render(){
        if (this.props.item.picUrl.length>0) {
        return(
            <TouchableOpacity onPress={this.props.itemClick}>
            <View>
               <Text style={itemStyles.title}> {this.props.item.description}
</Text>
                <View style={itemStyles.contentView}>
                    <Image style={itemStyles.image} source=
{{uri:this.props.item.picUrl}}/>
                    <Text style={itemStyles.subTitle}>
{this.props.item.title}</Text>
                </View>
                <Text style={itemStyles.time}> {this.props.item.ctime}
</Text>
                <View style={itemStyles.line}></View>
            </View>
            </TouchableOpacity>
        );
        }else{
        return(
            <TouchableOpacity onPress={this.props.itemClick}>
            <View>
               <Text  style={itemStyles.title}>{this.props.item.description}
</Text>
```

```
                    <Text style={itemStyles.detail}>{this.props.item.title}
</Text>
                    <Text style={itemStyles.time}>{this.props.item.ctime}
</Text>
                    <View style={itemStyles.line}></View>
                </View>
                </TouchableOpacity>
            );
            }
        }
```

修改 PageView 类的 contentView 方法：

```
        contentView(width){
            let views = new Array();
            for(let i=0;i<this.netTool.apis.length;i++){
                let element = (<FlatList key={i} style={{width:width}}
data={this.state.dataArray[i]}
                    renderItem={({item})=>{
                        return(<ItemView item={item} itemClick={()=>{
                                this.props.goDetail(item);
                            }}/>);
                    }}
                />);
                views.push(element);
            }
            return views;
        }
```

在 MainView 中引入 Navigator 组件：

```
    import { Navigator } from 'react-native-deprecated-custom-components';
```

将 MainView 类的实现修改如下：

```
    export default class MainView extends Component{
        render(){
            return(
                <Navigator ref='navigation' initialRoute={{title:'掌上新闻',
type:'root'}} renderScene={(route,navigator)=>{
                        if (route.type === 'root') {
                            return(
                                <View style={{flex:1}}>
                                    <NavigationBar />
                                    <TitleBar click={this.clickTitle}
ref="titleBar"/>
                                    <PageView ref='pageView' scrollEnd=
{this.scrollEnd} goDetail={(item)=>{
```

```
                                navigator.push({
                                    title:item.description,
                                    type:'detail',
                                    uri:item.url
                                });
                            }}/>
                        </View>
                    );
                }else if(route.type==='detail'){
                    return(
                        <View style={{flex:1}}>
                            <NavigationBar />
                            <DetailPage uri={route.uri}/>
                        </View>
                    );
                }
            }}
        />
    );
    }
    clickTitle=(index)=>{
        this.refs.navigation.refs.pageView.show(index);
    }
    scrollEnd = (index)=>{
        this.refs.navigation.refs.titleBar.selected(index);
    }
}
```

在上面的代码中，我们使用路由类型来进行页面跳转的控制。需要注意，对于 ref 的引用，我们需要借助 Navigator 实例进行传递。下面在 view 文件夹下新建一个 DetailPage.js 文件，实现代码如下：

```
import React, { Component } from 'react';
import {
    WebView
} from 'react-native';
export default class DetailPage extends Component{
    render(){
        let uri =  this.props.uri;
        return(
            <WebView source={{uri:uri}}/>
        );
    }
}
```

运行工程，单击目录中的某个新闻会跳转到相应的详情页。

11.8　完善下拉刷新与上拉加载更多功能

掌上新闻应用的下拉刷新和上拉加载功能的开发方法与微信热门文章相似。不同的是，掌上新闻应用更加复杂一些，新闻类目较多，在进行刷新和加载功能开发时，我们要注意找准执行功能的界面。

首先修改 PageView 类的构造方法：

```
    constructor(props){
        super(props);
        this.netTool =  new NetTool();
        let dataArray = new Array();
        let refreshArray = new Array();
        for (let i = 0; i < this.netTool.apis.length; i++) {
            dataArray.push([]);
            refreshArray.push({refreshing:false,page:1});
        }
        this.state={
            dataArray:dataArray,
            refreshArray:refreshArray
        }
        this.hasLoad = false;
        this.getData(0,1);
    }
```

上面的代码向 PageView 类的状态中添加了一个 refreshArray 数组，这个数组中的对象用来控制刷新状态以及加载的分页数。hasLoad 属性用来标记当前是否正在进行数据请求，这样做的目的是为了防止重复请求。

修改 PageView 类的 contentView 方法：

```
    contentView(width){
        let views = new Array();
        for(let i=0;i<this.netTool.apis.length;i++){
            let element = (<FlatList key={i} style={{width:width}}
data={this.state.dataArray[i]}
                renderItem={({item})=>{
                    return(<ItemView item={item} itemClick={()=>{
                            this.props.goDetail(item);
                        }}/>);
                }}
            refreshing={this.state.refreshArray[i].refreshing}
            onRefresh={()=>{
                if (this.hasLoad) {
```

```
                    return;
                   }
                    this.state.refreshArray[i].page=1;
                    this.state.refreshArray[i].refreshing=true;
                    this.setState({
                        refreshArray:this.state.refreshArray,
                        dataArray:this.state.dataArray
                    });
                    this.getData(i,1);
                }}
                onEndReached={()=>{
                    this.state.refreshArray[i].page+=1;
                    this.getData(i,this.state.refreshArray[i].page);
                }}
                onEndReachedThreshold={0.5}/>);
                views.push(element);
            }
            return views;
        }
```

修改获取数据的方法 getData：

```
        getData(index,page){
            if(this.hasLoad){
                return;
            }
            this.hasLoad = true;
            this.netTool.getNewsData(index,page,(data)=>{
                let list = data.newslist;
                let array = this.state.dataArray;
                let i = 20*(page-1);
                if (list) {
                    list.forEach((item)=>{
                        item.key = i++;
                    });
                }
                if (page==1) {
                    array[index] = list;
                }else{
                    array[index] = array[index].concat(list);
                }
                this.state.refreshArray[index].refreshing=false;
                this.setState({
                    dataArray:array,
                    refreshArray:this.state.refreshArray
                }
```

```
            );
            this.hasLoad = false;
        });
    }
```

运行工程，刷新和加载功能就开发完成了。注意，代码 let i = 20*(page-1)中的 20 需要与接口数据的返回数据条数保持一致，这是因为我们在写接口时将控制数据返回数量的参数设定为了 20。

11.9　完善导航栏

导航栏的作用除了用来显示当前界面的标题外，往往还会布局一些功能按钮，例如在详情页需要有返回按钮、与掌上新闻应用的收藏相关的功能按钮。

按照我们的设计，在首页用户就可以跳转到收藏夹页面进行往期收藏的浏览，在新闻详情页面可以进行当前新闻的收藏。在 NavagationBar.js 文件中编写如下代码：

```
import React, { Component } from 'react';
import {
  View,
  StyleSheet,
  Text,
  TouchableOpacity
} from 'react-native';
export default class NavigationBar extends Component{
    render(){
        if (this.props.type === 'root') {
            return(
            <View style={naviStyles.bar}>
                <View style={naviStyles.textView}>
                    <Text style={naviStyles.text}>
                        {this.props.title}
                    </Text>
                </View>
                <TouchableOpacity style={naviStyles.rightButton}>
                    <Text>收藏夹</Text>
                </TouchableOpacity>
            </View>
            );
        }else if(this.props.type === 'detail'){
            return(
            <View style={naviStyles.bar}>
                <View style={naviStyles.textView}>
                    <Text style={naviStyles.text}>
                        {this.props.title}
```

```
                        </Text>
                    </View>
                    <TouchableOpacity onPress={this.props.pop}
style={naviStyles.leftButton}>
                        <Text>返回</Text>
                    </TouchableOpacity>
                    <TouchableOpacity style={naviStyles.rightButton}>
                        <Text>添加收藏</Text>
                    </TouchableOpacity>
                </View>
                );
            }else if(this.props.tyoe === 'collection'){
            return(
                <View style={naviStyles.bar}>
                    <View style={naviStyles.textView}>
                        <Text style={naviStyles.text}>
                            {this.props.title}
                        </Text>
                    </View>
                    <TouchableOpacity onPress={this.props.pop}
style={naviStyles.leftButton}>
                        <Text>返回</Text>
                    </TouchableOpacity>
                </View>
              );
            }
        }
    }
    let naviStyles = StyleSheet.create({
        bar:{
            height:64,
            backgroundColor:'rgb(241,241,241)',
            flexDirection:'row',
        },
        textView:{
            alignSelf:'center',
            justifyContent:'center',
            flex:1
        },
        text:{
            textAlign:'center',
            fontSize:17
        },
```

```
        rightButton:{
            position:'absolute',
            right:15,
            alignSelf:'center',
        },
        leftButton:{
            position:'absolute',
            left:15,
            alignSelf:'center',
        },
    });
```

上面的代码通过判断不同的来源渲染不同的界面。收藏夹功能的相关代码这里并没有编写，后面章节再专门处理收藏功能。修改 MainView 类的 render 方法如下：

```
        render(){
            return(
                <Navigator ref='navigation' initialRoute={{title:'掌上新闻',
type:'root'}} renderScene={(route,navigator)=>{
                        if (route.type === 'root') {
                            return(
                                <View style={{flex:1}}>
                                    <NavigationBar title={route.title}
type={route.type}/>
                                    <TitleBar click={this.clickTitle}
ref="titleBar"/>
                                    <PageView ref='pageView'
scrollEnd={this.scrollEnd} goDetail={(item)=>{
                                        navigator.push({
                                            title:item.description,
                                            type:'detail',
                                            uri:item.url
                                        });
                                    }}/>
                                </View>
                            );
                        }else if(route.type==='detail'){
                            return(
                                <View style={{flex:1}}>
                                   <NavigationBar title={route.title}
type={route.type} pop={()=>{
                                    navigator.pop();
                                   }}/>
                                    <DetailPage uri={route.uri}/>
```

```
                        </View>
                    );
                }
            }}
        />
    );
}
```

运行工程，可以查看导航栏的开发效果。

11.10 添加收藏夹功能

要完成本节的内容，我们需要使用 React Native 中的数据持久化技术。首先在项目的根目录中创建一个命名为 Data 的文件夹，在其中创建一个命名为 DataManager.js 的文件，这个类用来进行收藏数据的管理，在其中编写如下代码：

```
import {
    AsyncStorage
} from 'react-native';
export default class DataManager {
    static getData(callback){
        AsyncStorage.getItem("data",(error,result)=>{
            if (!error) {
                result=result?result:"";
                let array = result.split('@@');
                callback(array);
            }
        });
    }
    static addCollection(url){
        this.getData((array)=>{
            array.push(url);
            let string = array.join('@@');
            AsyncStorage.setItem('data',string);
        });
    }
}
```

注意，DataManager 类中的方法都是使用的静态方法，不需要实例就可以调用。

完善 NavagationBar 的 render 方法：

```
    render(){
        if (this.props.type === 'root') {
            return(
```

```
                <View style={naviStyles.bar}>
                    <View style={naviStyles.textView}>
                        <Text style={naviStyles.text}>
                            {this.props.title}
                        </Text>
                    </View>
                    <TouchableOpacity onPress={this.props.goCollection}
style={naviStyles.rightButton}>
                        <Text>收藏夹</Text>
                    </TouchableOpacity>
                </View>
                );
            }else if(this.props.type === 'detail'){
                return(
                <View style={naviStyles.bar}>
                    <View style={naviStyles.textView}>
                        <Text style={naviStyles.text}>
                            {this.props.title}
                        </Text>
                    </View>
                    <TouchableOpacity onPress={this.props.pop}
style={naviStyles.leftButton}>
                        <Text>返回</Text>
                    </TouchableOpacity>
                    <TouchableOpacity onPress={()=>{
                        DataManager.addCollection(this.props.url);
                        Alert.alert("提示","您已经添加到收藏夹");
                    }} style={naviStyles.rightButton}>
                        <Text>添加收藏</Text>
                    </TouchableOpacity>
                </View>
                );
            }else if(this.props.type === 'collection'){
                return(
                <View style={naviStyles.bar}>
                    <View style={naviStyles.textView}>
                        <Text style={naviStyles.text}>
                            {this.props.title}
                        </Text>
                    </View>
                    <TouchableOpacity onPress={this.props.pop}
style={naviStyles.leftButton}>
                        <Text>返回</Text>
```

```
                    </TouchableOpacity>
                </View>
            );
        }
    }
```

在 View 文件夹下创建一个命名为 CollectionView.js 的文件，用来展示收藏列表，在其中编写如下代码：

```
    import React, { Component } from 'react';
    import {
        View,
        FlatList,
        TouchableOpacity,
        Text,
        StyleSheet
    } from 'react-native';
    import DataManager from './../Data/DataManager';
    export default class CollectionView extends Component{
        constructor(props){
            super(props);
            let data = new Array();
            this.state={
                dataArray:new Array()
            };
            DataManager.getData((array)=>{
                for(let i=0;i<array.length;i++){
                    if (array[i].length===0) {
                        continue;
                    }
                    data.push({key:i,url:array[i]});
                    this.setState({
                        dataArray:data,
                    });
                }
            });
        }
        render(){
            return(
                <View style={{flex:1,backgroundColor:'white'}}>
                    <FlatList style={collectionStyles.list}
data={this.state.dataArray}
                    renderItem={({item})=>{
                        return(
                            <TouchableOpacity onPress={()=>{
                                this.props.goDetail(item.url);
```

```
                            }}>
                                <View>
                                <Text style={collectionStyles.text}>
{item.url}</Text>
                                <View style={collectionStyles.line}></View>
                                </View>
                            </TouchableOpacity>
                        );
                    }}/>
                </View>
            );
        }
    }
    let collectionStyles = StyleSheet.create({
        text:{
            fontSize:15,
            marginLeft:15,
            marginRight:15,
            marginBottom:10,
            marginTop:10
        },
        line:{
            marginLeft:15,
            height:1,
            backgroundColor:'rgb(111,111,111)',
            marginBottom:10
        }
    });
```

最后，修改 MainView 类的 render 方法，添加路由跳转：

```
        render(){
            return(
                <Navigator ref='navigation' initialRoute={{title:'掌上新闻',
type:'root'}} renderScene={(route,navigator)=>{
                        if (route.type === 'root') {
                            return(
                                <View style={{flex:1}}>
                                    <NavigationBar title={route.title}
type={route.type}  goCollection={()=>{
                                        navigator.push({
                                            title:"收藏夹",
                                            type:'collection'
                                        });
                                    }}/>
```

```
                                    <TitleBar click={this.clickTitle}
ref="titleBar"/>
                                    <PageView ref='pageView'
scrollEnd={this.scrollEnd} goDetail={(item)=>{
                                        navigator.push({
                                            title:item.description,
                                            type:'detail',
                                            uri:item.url
                                        });
                                    }}/>
                                </View>
                            );
                        }else if(route.type==='detail'){
                            return(
                                <View style={{flex:1}}>
                                    <NavigationBar title={route.title}
type={route.type} pop={()=>{
                                    navigator.pop();
                                    }} url={route.uri}/>
                                    <DetailPage uri={route.uri}/>
                                </View>
                            );
                        }else if(route.type==='collection'){
                            return(
                            <View style={{flex:1}}>
                                    <NavigationBar title={route.title}
type={route.type} pop={()=>{
                                    navigator.pop();
                                    }}/>
                                    <CollectionView goDetail={(url)=>{
                                        navigator.push({
                                            title:"来自收藏夹",
                                            type:'detail',
                                            uri:url
                                        });
                                    }}/>
                            </View>
                            );
                        }
                    }}
                />
            );
        }
```

运行工程，收藏新闻后，可以直接从收藏列表跳转到新闻详情页。

11.11　优化方向与应用图标设置

到 11.10 节为止，我们已经将掌上新闻应用程序的所有核心功能完成，但是这个应用程序并不完美，还有许多细节需要优化，这里列出几点供读者参考：

（1）在对新闻进行收藏时，并没有进行是否已经收藏的判断，后面可以加上相关逻辑，如果此新闻已经被收藏，则不显示收藏按钮或者用户单击收藏按钮后提示已经收藏。

（2）收藏功能目前没有取消逻辑，用户收藏的数据会越来越多，应该为用户添加一个取消收藏的功能。

（3）目前的收藏功能很简单，收藏列表界面显示的都是 URL 地址，优化这一部分逻辑，使其显示标题并支持新闻分类。

除了上面列出的，还有很多细节可以改进，读者可以自由发挥，应用前面学习的 React Native 知识。

一个完整的商业应用程序还需要有一个应用图标，如果没有配置，在 iOS 平台与安卓平台上的应用图标分别如图 11-6 与图 11-7 所示。



图 11-6　iOS 平台上的默认图标

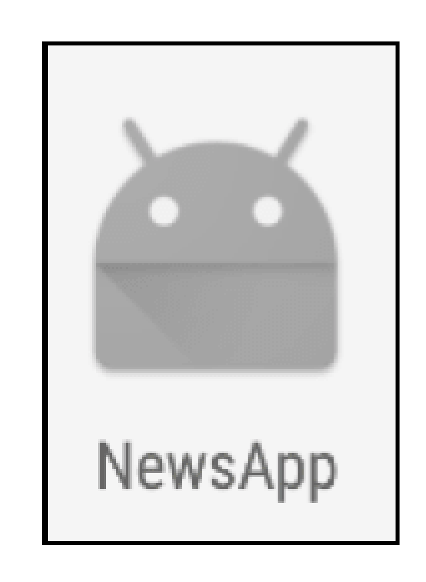


图 11-7　安卓平台上的默认图标

配置应用程序图标并不复杂，对于 iOS 平台，可以直接打开目录中的 iOS 工程文件，如图 11-8 所示。

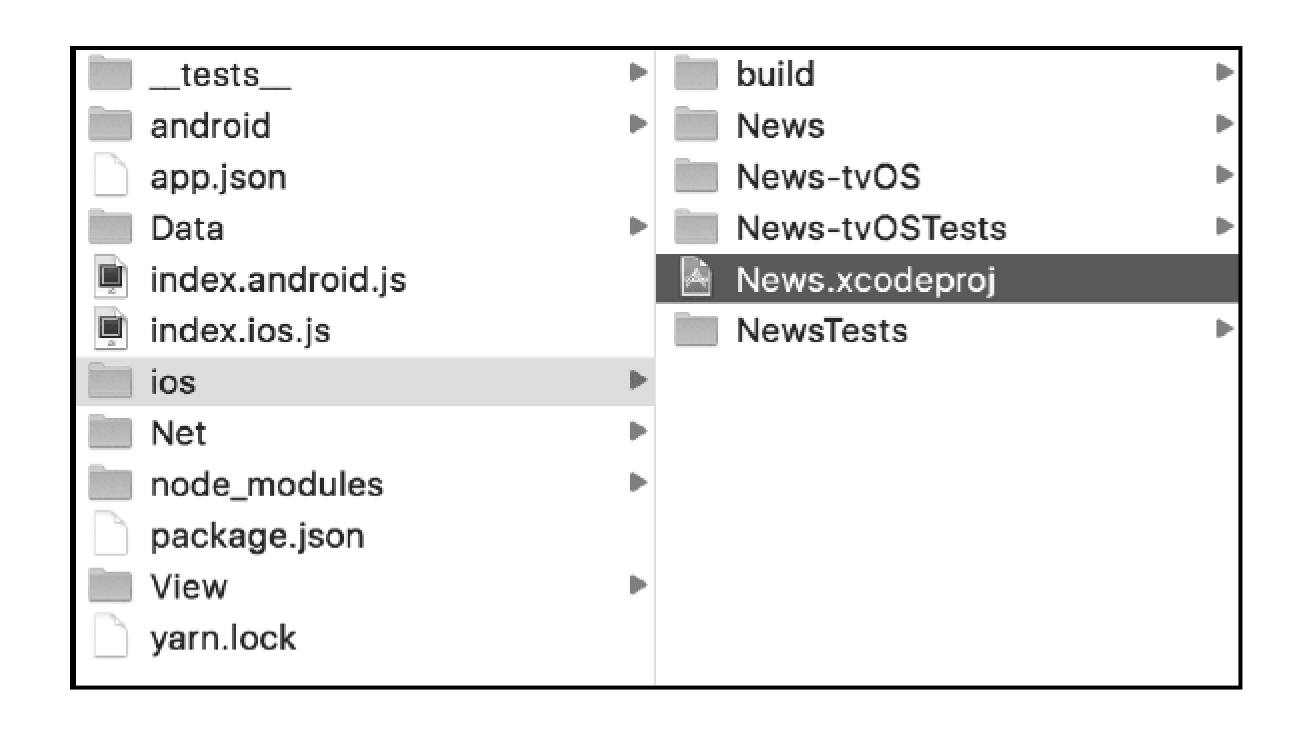


图 11-8　打开 iOS 工程文件

选择工程文件中的 images.xcassets 素材包，将对应尺寸的应用程序图标放入指定的位置即可，如图 11-9 所示。

对于 Android 工程，找到如下路径中的素材文件夹（见图 11-10）：

android→app→src→main→res

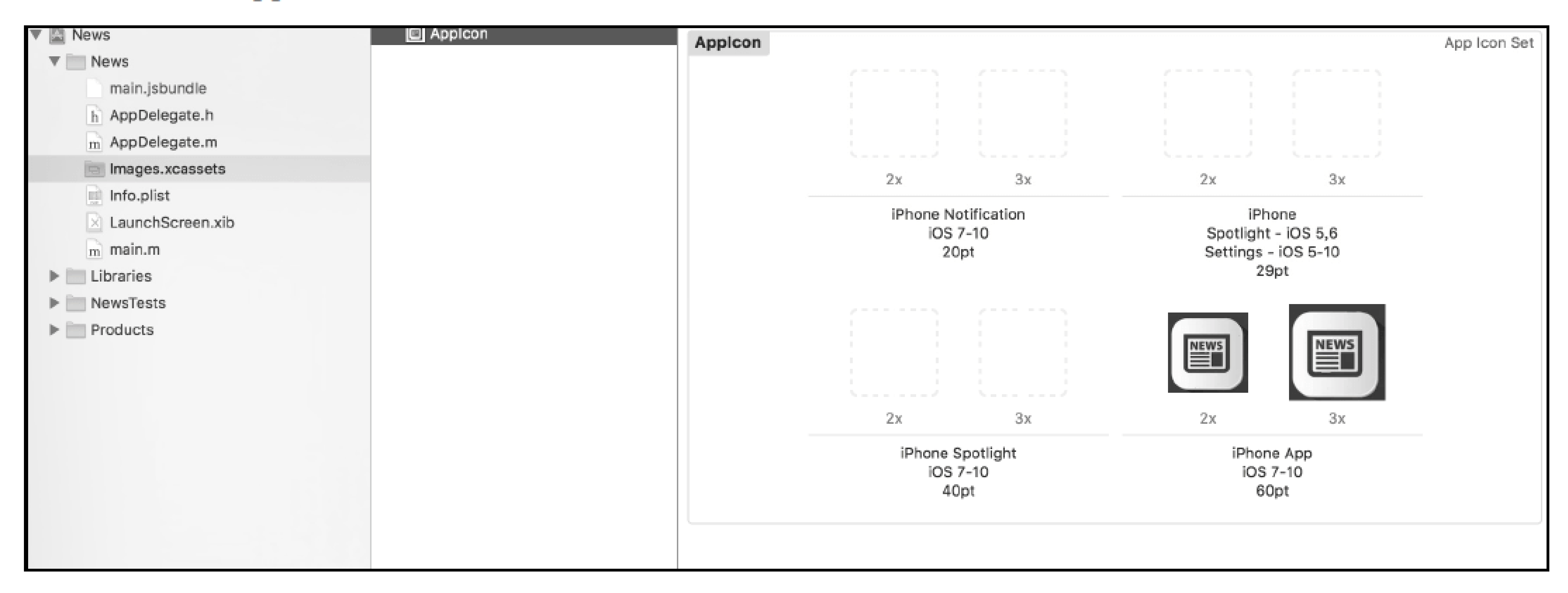


图 11-9　配置 iOS 应用程序图标

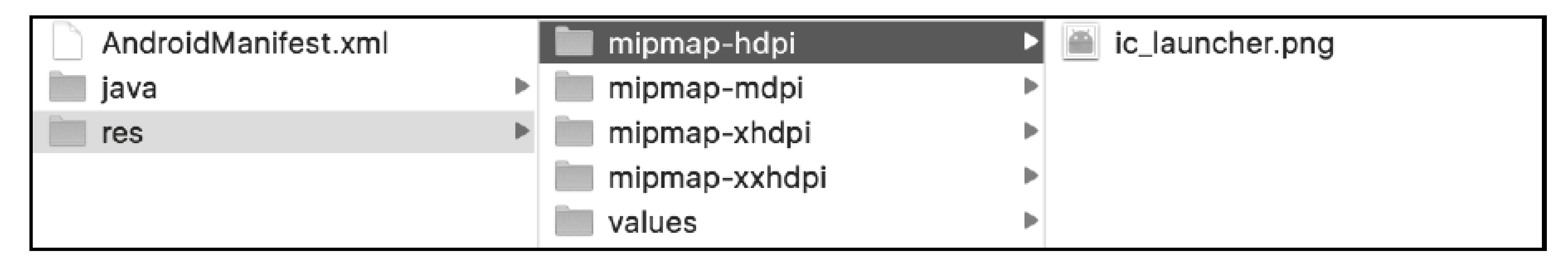


图 11-10　存放 Android 素材的文件夹

分别替换各个文件夹中的 ic_launcher.png 文件即可。需要注意的是，图片尺寸和名称要保持一致。重新编译工程，应用程序的图标就变成了我们所设置的图标。

第 12 章

React Native 高级技巧

如果你已经顺利地学习到了本章内容，那么相信你的 React Native 开发水平已经到达了一定程度，使用 React Native 开发完整的原生应用程序已经不成问题，但是在 React Native 学习之路上，你所掌握的知识还远远不够，你能够开发完整的应用但却不一定可以将应用的性能优化到最优；你能灵活运用当前版本的 React Native 来做需求开发却不一定知道如何对旧项目的 React Native 进行升级与迁移；你会通过 Chrome 开发者工具来对 React Native 工程进行调试却没有使用过更多强大的 React Native 调试工具。除了上面所提到的这些，如何在原生模块中使用 React Native 和如何封装自己的 React Native 组件也是十分重要的，本章将介绍这些内容并起到投石问路的作用，之后你还需要不断地通过官方文档等资料更深入地理解 React Native。

12.1 直接操作组件的属性

在对组件的属性进行更新时，前面我们一直使用更新“状态”的方式来实现。这也是 React Native 中推荐开发者使用的更新界面的方法。但是使用“状态”来进行组件的更新有一个致命的缺陷，其会刷新整个组件，即重新调用组件的 render 方法，对于一些复杂的自定义组件，如果仅仅为了更新其中的某一部分，刷新状态会造成很大的性能浪费，如果要自己实现动画，这也是造成界面卡顿和掉帧的主要原因。

虽然通过“状态”来管理你的应用显示逻辑是十分优秀的设计方案，但是你依然需要学会怎么直接来操作组件的属性。我们先来看下面的例子。

在 HelloWorld 工程的 Demo 文件夹下新建一个命名为 NativeDemo.js 的文件，在其中编写如下代码：

```
import React, { Component } from 'react';
import {View,Text} from 'react-native';
```

```
    export default class NativeDemo extends Component{
        constructor(props){
            super(props);
            this.state={
                color:'red'
            }
        }
        render(){
            console.log("render");
            return(
                <View>
                    <Text onPress={()=>{
                        this.setState({
                            color:'green'
                        });
                    }} style={{top:200,fontSize:30,textAlign:"center",
color:this.state.color}} ref="text">{this.props.text}</Text>
                </View>
            );
        }
    }
```

上面的代码中，我们在 render 方法里添加了一个打印语句，运行项目，在 Chrome 浏览器的开发者工具中可以看到，每次单击便签，实际上都会重新执行 render 方法，如图 12-1 所示。



图 12-1　多次调用 render 方法

将 render 方法做如下修改：

```
        render(){
            console.log("render");
            return(
                <View>
                    <Text onPress={()=>{
                        this.refs.text.setNativeProps({
                                style:{color:'green'}
                            });
                    }} style={{top:200,fontSize:30,textAlign:"center",
color:this.state.color}} ref="text">{this.props.text}</Text>
                </View>
            );
        }
```

setNativeProps 方法可以直接操作原生组件的属性，如上代码所示，当单击按钮时直接修改了 Text 组件的文字颜色，并且不会重新调用 render 方法，这样就实现了自定义组件的局部刷新。

抽 丝 剥 茧

虽然这种局部更新组件的方法十分好用，但是除非必要，建议最好还是采用更新“状态”来刷新组件，使用“状态”能使代码调理更加清晰，更易于进行界面与数据间的管理。

12.2 对 React Native 版本进行升级

React Native 如此流行的原因不仅仅因为其是一个十分优秀的移动端跨平台方案，更重要的是 Facebook 团队一直保持着对它的更新和维护。React Native 框架的版本也一直在升级。将旧版本的工程升级为新版本也是开发者时常要做的一件事情。

在工程目录下使用如下命令可以检查当前的 React Native 版本：

```
react-native --version
```

在终端执行上面的命令后，效果如图 12-2 所示。

```
vipdeMacBook-Pro-4:HelloWorld jaki$ react-native --version
react-native-cli: 2.0.1
react-native: 0.44.0
vipdeMacBook-Pro-4:HelloWorld jaki$
```

图 12-2 检查当前 React Native 版本

所谓进行 React Native 版本的升级，实质上就是使用框架的新版本的代码来代替旧版本的，一种笨方法是将所有工程中你自己添加的文件和代码复制出来，然后新建一个工程，将这些文件和代码还原回去，但这种升级方式一般不会被开发者所选择，我们通常会采用 React Native 自动升级工具完成。

首先需要安装 react-native-git-upgrade 工具，使用如下命令进行安装：

```
npm install -g react-native-git-upgrade
```

如果想让 React Native 工程升级到最新版本，可以在工程根目录下直接执行如下命令：

```
react-native-git-upgrade
```

当然如果你想升级到指定版本也是可以的，使用如下命令并指定版本号：

```
react-native-git-upgrade X.Y.Z
```

一般情况下，如果你没有修改过框架文件，更新完成后是不会产生异常或冲突的，如果有冲突产生，需要你根据自己的实际情况来解决这些冲突（实际上就是选择框架的代码或者是你自己的代码）。

抽 丝 剥 茧

如果工程中 React Native 版本与要更新到的版本相差极大，建议你一定要对代码进行备份。

12.3 React Native 的更多调试技巧

使用 Chrome 开发者工具可以实时查看运行中 React Native 应用的打印信息、网络情况等，并且可以添加断点进行工程代码的调试。熟练使用 Chrome 开发者工具就已经可以帮助你完成大部分调试工作。实际上，React Native 中还内置了许多有用的调试工具。

在 iOS 模拟器上可以使用 command+D（真机是摇晃手机）调出开发者菜单，在 Android 模拟器上使用 command+M（真机是摇晃手机）调出开发者菜单。这个操作你并不陌生，要使用 Chrome 浏览器进行代码的远程调试就是在这个开发者菜单中设置的。你也一定发现了，这个开发者菜单中还提供了许多其他的功能，如图 12-3 所示。

Reload 用来进行界面的手动刷新，这个刷新方式会重新加载远程的 JavaScript 文件。与刷新相关的功能还有 Live Reload 和 Hot Reloading 两种，如果开启了 Live Reload 功能，当你编写的 JavaScript 文件有变化时，React Native 会自动进行 Reload 刷新操作，省去了开发者经常要手动刷新的烦恼，Hot Reloading 功能则更加强大，其不会重新加载应用，只会增量地刷新界面，也就是说，如果你在应用运行中修改了界面某个元素的颜色代码，此修改会即刻同步到界面上。

JS Remotely 功能用来开启远程调试，前面我们已经介绍过很多，这里就不再重复了。

Start Systrace 功能用来开启性能分析，开启后，开发者菜单中的此选项会变成 Stop Systrace。性能分析结束后会生成一份分析报告给开发者。

Toggle Inspector 功能十分强大，可以实时显示界面上元素的风格设置、应用的网络访问情况等，如图 12-4 所示。

Perf Monitor 功能用来显示应用的状态信息，例如 UI 线程帧率、JS 线程帧率、函数运行耗时和内存使用情况等，如图 12-5 所示。

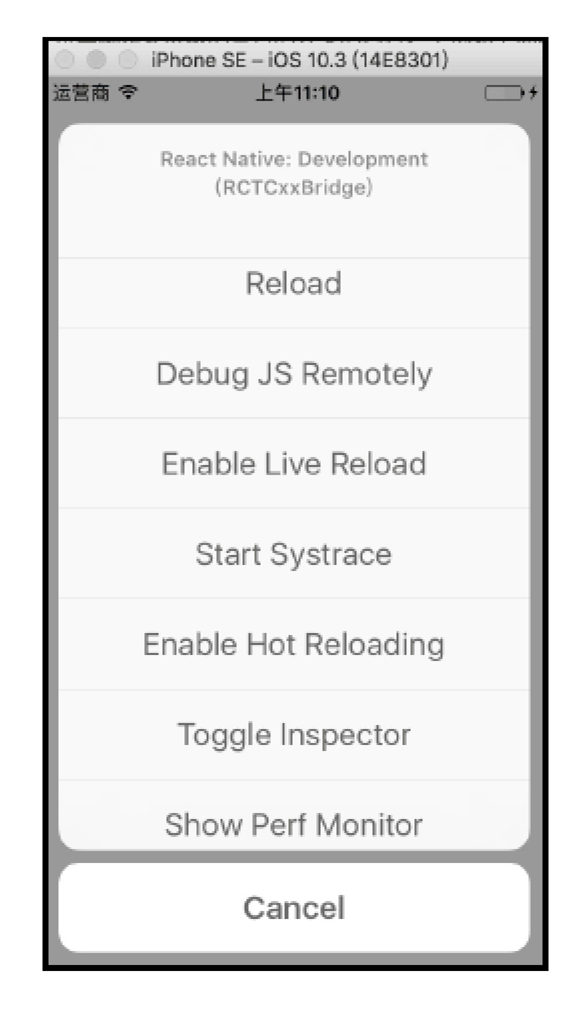




图 12-3 开发者菜单

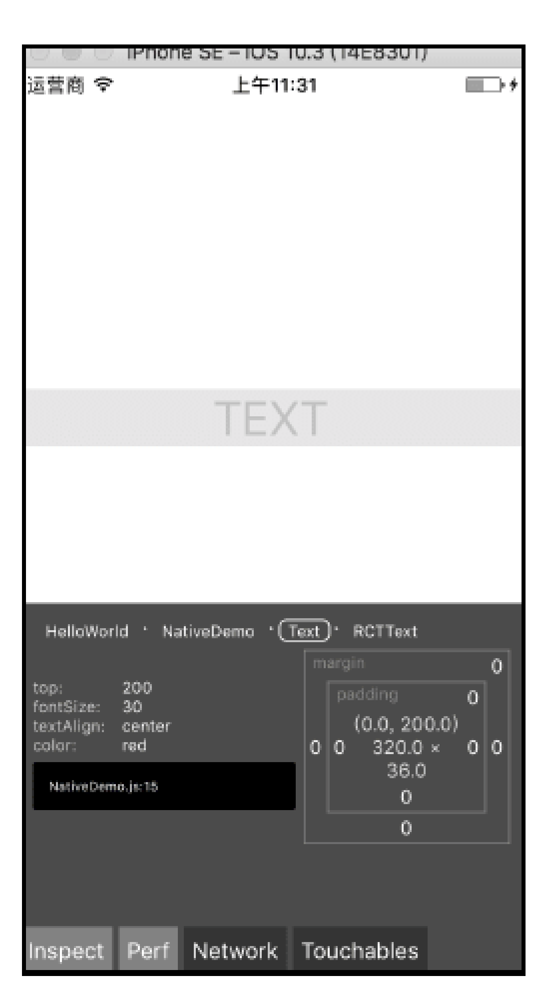


图 12-4 界面监测调试功能

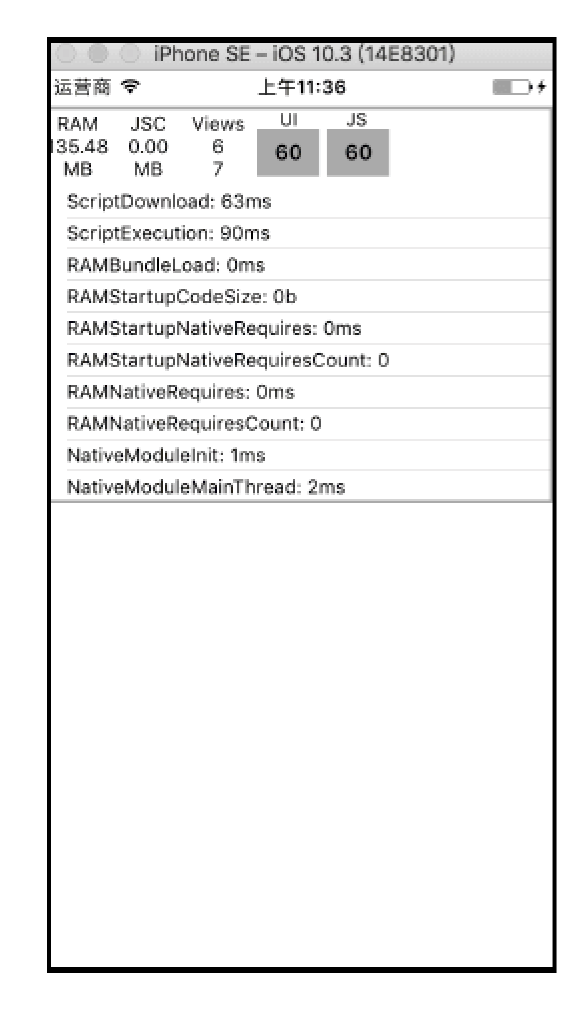


图 12-5 监测应用状态与函数耗时

上面这些工具在一般情况下并不一定会用到，但是掌握它们后一定会在优化项目时助你一臂之力。

12.4　React Native 插件开发

我们使用 React Native 框架完成一般的项目当然不在话下，但 React Native 框架并不是万能的，它并没有将原生模块的功能完完整整地实现。又或者说，在做原生开发时我们通常会使用一些第三方库和插件，这些自然是 React Native 中不包含的功能。本节我们就一起来看看 React Native 更加强大的一面，它支持你构建自定义的原生模块并在工程中使用 JavaScript 来调用这些模块。在学习本节前，建议你最好积累一些原生开发的经验，否则本节的内容可能会比较难理解。

12.4.1　构建 iOS 工程的原生模块

本小节我们通过一个简单的实例演示如何构建自定义的原生模块，以及如何在 JavaScript 工程中调用这些模块。首先，打开 HelloWorld 工程下 IOS 文件夹中的 IOS 工程文件，如图 12-6 所示。

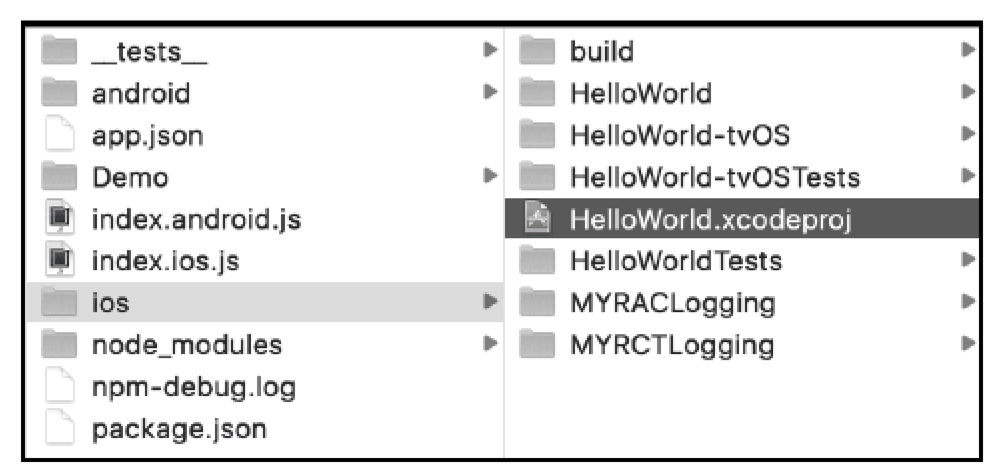


图 12-6　打开 ios 工程文件

如果你没做过原生的 iOS 开发，可能对 Xcode 开发工具还不太熟悉，这里我向你简单介绍一下。

Xcode 工具的开发界面可以分为 5 个部分，如图 12-7 所示。其中，导航区用来展示文件列表、检索列表、警告和错误信息、内存信息等；编码区进行代码的编写；调试与输出区用来进行代码断点的调试以及信息的输出；配置区可以对文件属性进行配置；资源区则是一些代码块、静态资源的列表。再看左侧的导航区，React Native 默认生成的工程中有一个 Libraries 组，这个组中就是 React Native 中内置的一些模块，我们可以仿照它来创建自己的模块。



图 12-7　Xcode 工具开发面板

在 Xcode 工具栏中选择 File→New→Project...选项，新建一个工程，如图 12-8 所示。

图 12-8　创建一个新工程

在弹出的窗口中选择 iOS 目录下的 Cocoa Touch Static Library 静态库工程，之后单击 Next 按钮，如图 12-9 所示。

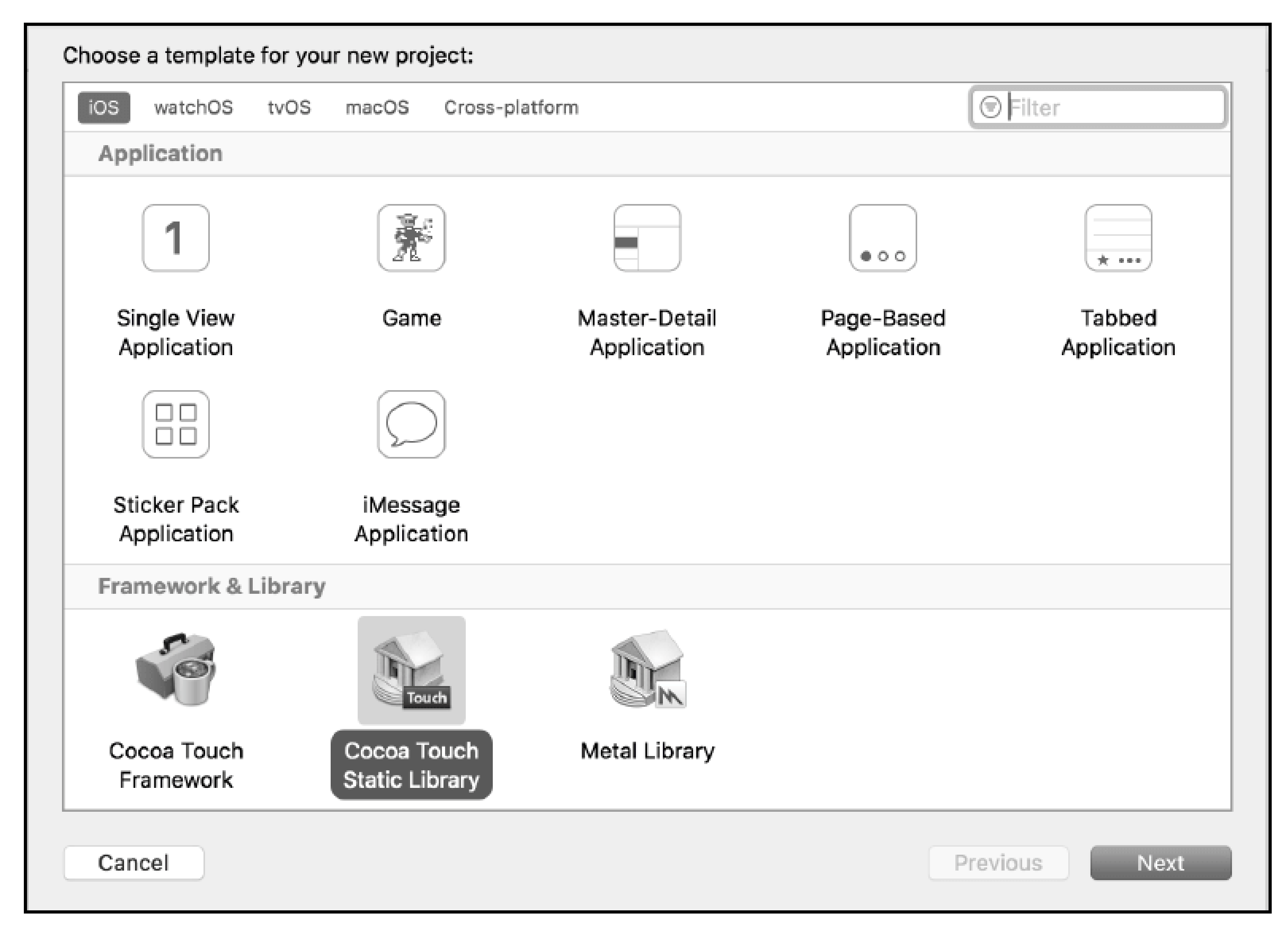


图 12-9　创建静态库

将工程命名为 RNTools（也可以是任意你喜欢的名称），直接单击 Next 按钮，如图 12-10 所示。

在弹出的配置窗口中，将此工程添加进 HelloWorld 工程中，并且将其加入 Libraries 组中，如图 12-11 所示。需要注意，如果你的窗口中没有这些选项，可以单击 Options 按钮唤出。

完成上面的工作后，HelloWorld 工程的 Libraries 组下会多一个 RNTools 工程。下面我们需要完成静态库的链接，过程如图 12-12 所示。

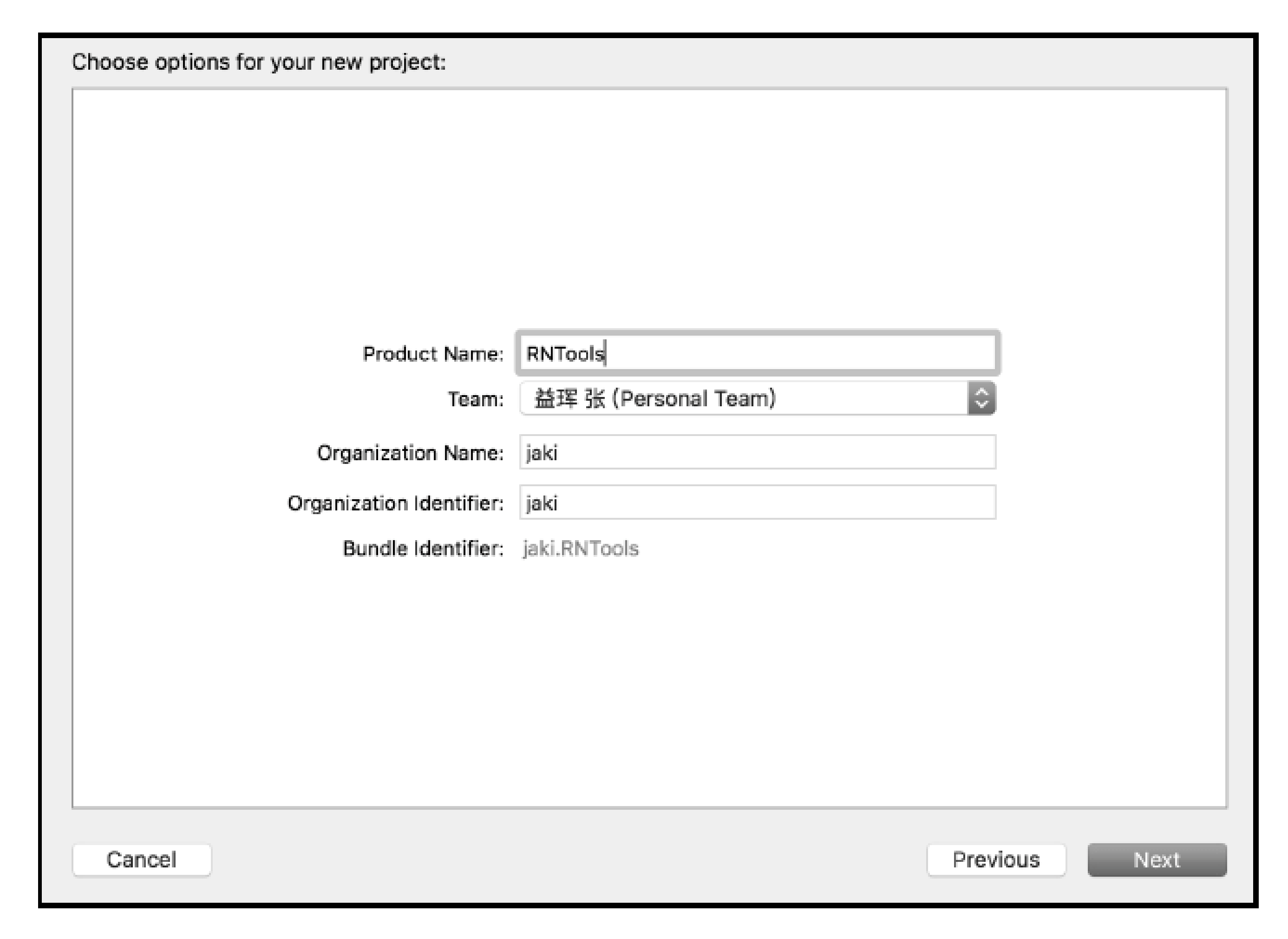


图 12-10　进行工程的命名

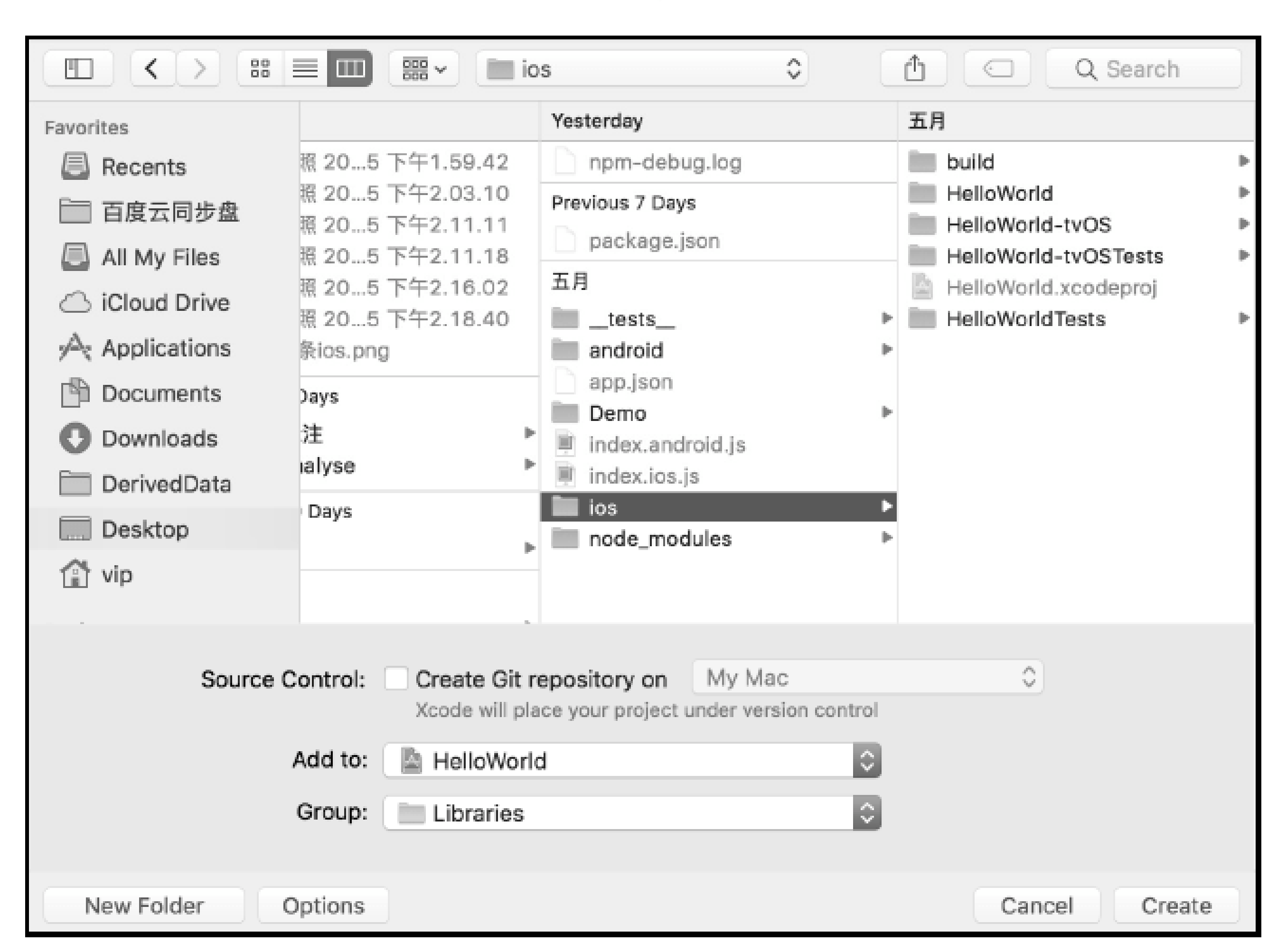


图 12-11　进行工程组配置



图 12-12　添加链接静态库

如图 12-12 的操作会唤出一个静态库列表，找到其中的 libRNTools.a 并添加，这个静态库就是我们刚刚创建的 RNTools 模块。

打开 RNTools 工程下的 RNTools.h 文件，要将一个原生类作为 React Native 的一个模块，需要在其中实现 RCTBridgeModule 协议的相关方法，首先在 RNTools.h 中添加如下头文件：

```
#import <Foundation/Foundation.h>
#import <React/RCTBridgeModule.h>
#import <React/RCTLog.h>
```

其中，Foundation 库为 Objective-C 语言的基础库，RCTBridgeModule 为 React Native 模块协议，RCTLog 用来进行 React Native 中的打印操作。让 RNTools 类遵守 RCTBridgeModule 协议，代码　如下：

```
@interface RNTools : NSObject<RCTBridgeModule>
@end
```

在 RNTools.m 文件中实现如下的宏来进行模块的导出，传入参数为模块的名称（即 JavaScript 调用原生模块的名称）：

```
RCT_EXPORT_MODULE(RNToolsManager);
```

抽 丝 剥 茧

如果 RCT_EXPORT_MODULE()宏中不传入任何参数，则默认会将当前类的名称作为模块名称。

使用 RCT_REMAP_METHOD 宏来进行原生函数的导出，在 RNTools 类中编写如下代码：

```
RCT_REMAP_METHOD(printf,param:(NSString *)param param2:(NSString *)param2)
{
    RCTLogInfo(@"这里是原生模块的 Log:参数 1-%@，参数 2-%@|来自类:%@",
param,param2,[self class]);
}
```

RCT_REMAP_METHOD 宏中的第一个参数为设置导出的方法名，这里我们取名为 printf，第二个参数为导出方法的参数列表。我们姑且实现这一个方法，重新编译工程，一个简单的 React Native 原生模块就构建完成了。

下面我们来实现 JavaScript 代码，在 Demo 文件夹下新建一个命名为 ToolsManager.js 的文件，在其中编写如下代码：

```
import { NativeModules } from 'react-native';
export var ToolsManager = NativeModules.RNToolsManager;
```

借助我们前面编写的 NativeDemo 示例，在 NativeDemo.js 中引入 ToolsManager 模块：

```
import {ToolsManager} from './ToolsManager';
```

在 NativeDemo 类的 render 方法中添加如下代码：

```
ToolsManager.printf("Jaki","珲少");
```

上面我们调用了原生模块的 printf 方法，并传入了两个参数，运行工程，打开调试工具，可以看到如图 12-13 所示的打印信息。

```
这里是原生模块的Log:参数1-Jaki，参数2-珲少|来自类:RNTools          RCTLog.js:43
```

图 12-13　调用原生模块方法

有时，可能需要定义一些原生的常量来供 JavaScript 访问，实现如下方法即可：

```
- (NSDictionary *)constantsToExport
{
  return @{ @"constName": @"constValue" };
}
```

到此，你已经掌握在 iOS 平台开发 React Native 原生模块的整体过程。如果你在 Android 平台上运行上面的工程就会抛出异常，下一小节我们就来看看如何开发 Android 平台的 React Native 原生模块。

12.4.2　构建 Android 工程的原生模块

在 Android 工程中自定义原生模块与在 iOS 工程中自定义原生模块的过程基本一致，但由于平台的本身差异，我们还是有必要专门用一小节来介绍如何在 Android 平台上定义原生模块，在 Android 平台定义原生模块需要进行如下几个步骤：

步骤 01 创建原生模块。

步骤 02 进行原生模块的包装。

步骤 03 将包装后的原生模块进行注册。

如 12.4.1 小节所说，你在 iOS 平台上运行工程时非常完美，Android 平台上却会抛出异常，我们需要在 Android 工程中也实现一个 RNToolsManager 模块。首先使用 Android Studio 工具打开项目的 Android 工程，即 HelloWorld 工程文件夹下面的 Android 文件夹。将项目的工程组织结构选择为 Android，如图 12-14 所示。

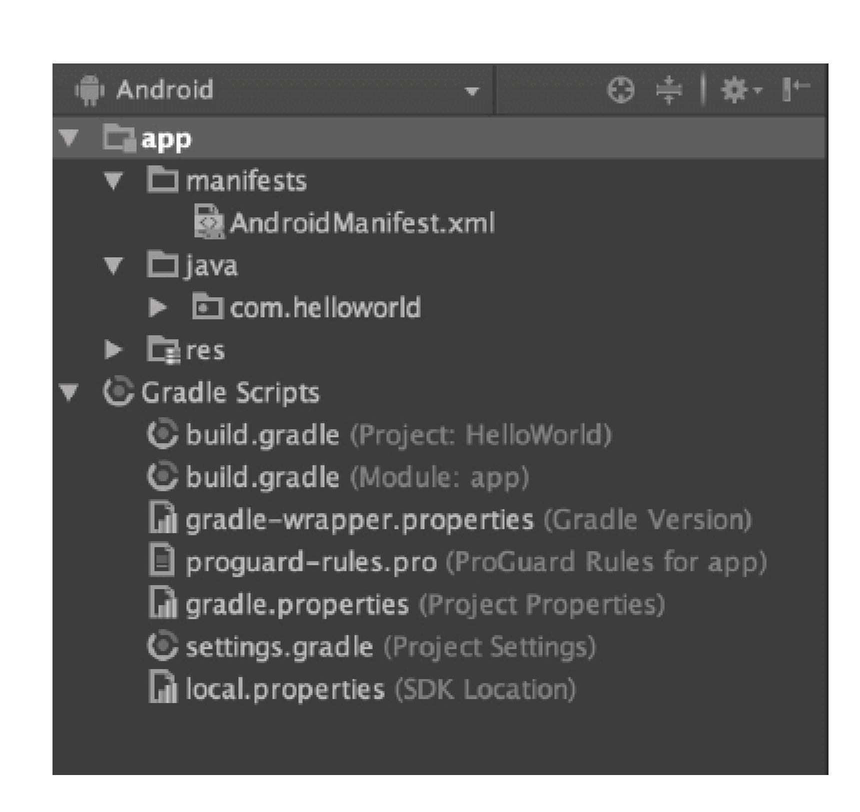


图 12-14　将工程的组织结构选择为 Android

图 12-14 中的 app→java→com.helloworld 包就是你的原生代码存放的位置。在 com.helloworld 包下新建一个 Java 类，将其命名为 RNToolsManager。Android 中的原生模块其实就是一个继承自 ReactContextBaseJavaModule 的类，在其中编写如下代码：

```
package com.helloworld;
import android.widget.Toast;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;

public class RNToolsManager extends ReactContextBaseJavaModule {
```

```
        public RNToolsManager(ReactApplicationContext reactContext) {
            super(reactContext);
        }
        @Override
        public String getName() {
            return "RNToolsManager";
        }
        @ReactMethod
        public void printf(String param1,String param2){
            Toast.makeText(getReactApplicationContext(), "andorid:"
+param1+param2,Toast.LENGTH_LONG).show();
        }
    }
```

其中，构造方法让 Android Studio 自动生成，不需要做任何修改。getName 方法用来设置原生模块的名称，要为原生模块添加方法需要使用@ReactMethod 注解的方式，这点和 iOS 工程有些区别，这里我们为了便于看到效果，弹出了一个简单的 Android 原生 Toast 提示。

完成了上面的工作，我们还需要对创建的 RNToolsManager 类进行包装，在 com.helloworld 包下再新建一个 Java 类，将其命名为 RNToolsManagerPackage，实现一个名为 ReactPackage 的接口，在其中编写如下代码：

```
    package com.helloworld;
    import com.facebook.react.ReactPackage;
    import com.facebook.react.bridge.NativeModule;
    import com.facebook.react.bridge.ReactApplicationContext;
    import com.facebook.react.uimanager.ViewManager;
    import java.util.ArrayList;
    import java.util.Collections;
    import java.util.List;

    public class RNToolsManagerPackage implements ReactPackage {
        @Override
        public List<NativeModule> createNativeModules(ReactApplicationContext
reactContext) {
            List<NativeModule> list = new ArrayList<>();
            list.add(new RNToolsManager(reactContext));
            return list;
        }
        @Override
        public List<ViewManager> createViewManagers(ReactApplicationContext
reactContext) {
            return Collections.emptyList();
        }
    }
```

之后，需要在 MainApplication.java 文件中进行 RNToolsManagerPackage 模块的注册，在 MainApplication 类的 getPackages 方法中添加这个包装过的模块，代码如下：

```
@Override
protected List<ReactPackage> getPackages() {
  return Arrays.<ReactPackage>asList(
      new MainReactPackage(),new RNToolsManagerPackage()
  );
}
```

要导出常量，需要实现如下方法：

```
@Override
public Map<String, Object> getConstants() {
  final Map<String, Object> constants = new HashMap<>();
  constants.put("constName",constValue);
  return constants;
}
```

到此，Android 平台上也可以支持 ToolsManager 的 printf 方法了，重新编译工程，在 Android 模拟器上运行，效果如图 12-15 所示。

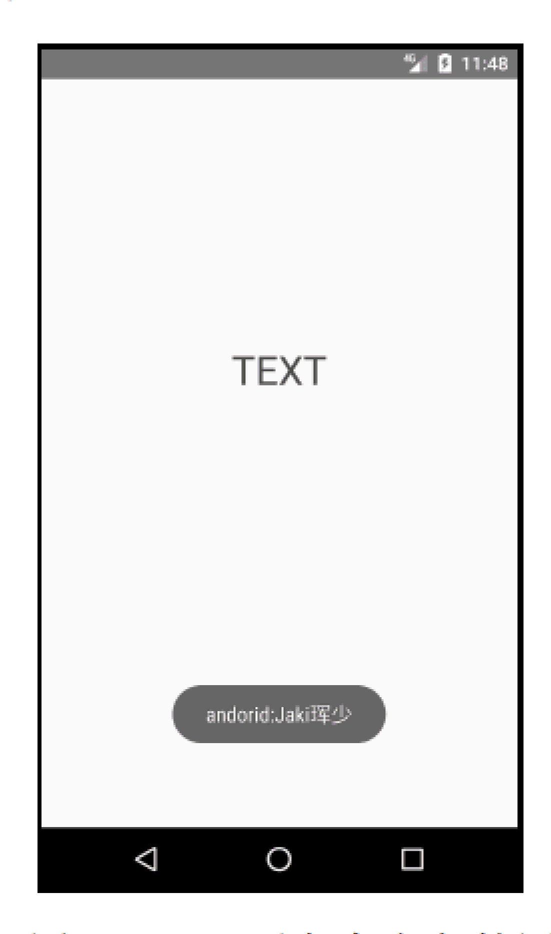


图 12-15　调用 Android 平台自定义的原生模块方法

12.4.3　深入了解原生模块的函数参数

从原生模块导出的方法无论是 iOS 平台还是 Android 平台，都不可以有返回值，但是我们可以通过回调函数的方法来进行传值交互。原生模块所支持函数的参数类型如表 12-1 所示。

表 12-1　原生模块支持的函数参数类型

JavaScript 参数类型	iOS 平台对应参数类型	Android 平台对应参数类型
String	NSString	String
Number	各种数值类型，如 NSInteger、float、double、CGFloat、NSNumber	各种数值类型，如 Integer、Double、Float

（续表）

JavaScript 参数类型	iOS 平台对应参数类型	Android 平台对应参数类型
Boolean	BOOL 或 NSNumber	Bool
Array	NSArray	ReadableArray
Object	NSDictionaty	ReadableMap
Function	RCTResponseSenderBlock	CallBack

回调函数是 JavaScript 代码与原生模块交互的一种重要方式，例如我们优化一下 12.4.2 小节构建的原生模块方法，让其拼接完整的字符串后传递回 JavaScript 代码中，修改 iOS 工程中的方法：

```
    RCT_REMAP_METHOD(printf,param:(NSString *)param param2:(NSString *)param2
param3:(RCTResponseSenderBlock)claaBack)
    {
        NSString * string = [NSString stringWithFormat:@"这里是原生模块的 Log:参数
1-%@，参数 2-%@|来自类:JavaScript",param,param2];
        claaBack(@[string]);
    }
```

修改 Android 工程中的方法：

```
        @ReactMethod
        public void printf(String param1, String param2, Callback callback){
            Toast.makeText(getReactApplicationContext(),"andorid:"+param1+param2,
Toast.LENGTH_LONG).show();
            callback.invoke("andorid:"+param1+param2);
        }
```

修改 NativeDemo.js 文件中原生模块的调用方法，为其添加一个回调函数：

```
            ToolsManager.printf("Jaki","珲少",(str)=>{
                console.log(str);
            });
```

重新编译运行工程，可以看到从原生模块传递回来的数据。需要注意，回调函数中传递的参数必须是数组，数组中的元素会被直接映射为 JavaScript 回调中的参数。

回调函数通常用来进行异步编程，你一定还记得，ES6 中的 Promise 对象也是异步编程的一种方案，在原生模块中也可以使用这种方式来与 JavaScript 进行交互，在 iOS 原生模块中导出一个新的方法，代码如下：

```
    RCT_REMAP_METHOD(println,success:(BOOL)success resolve:
(RCTPromiseResolveBlock)resolve reject:(RCTPromiseRejectBlock)reject)
    {
        if (success) {
            resolve(@[@"成功"]);
        }else{
            //三个参数，分别为状态码、异常信息和错误对象
```

```
            reject(0,@"失败",nil);
        }
    }
```

在 Android 的原生模块中导出新方法，代码如下：

```
    @ReactMethod
    public void println(Boolean success, Promise promise){
        if (success){
            promise.resolve(("成功"));
        }else{
            promise.reject("0","失败");
        }
    }
```

修改 JavaScript 代码：

```
        var promise = ToolsManager.println(true);
        promise.then((value)=>{
            console.log(value);
        },(error)=>{
            console.log(error);
        });
```

这种使用 Promise 进行异步编程的方式有着同步编程的风格，代码结构清晰，书写方便。

12.5　封装原生 UI 组件

所谓 UI（User Interface），即用户接口，通俗来讲就是我们一直使用的界面组件。通过 12.4 节的学习，我相信你能够构建自己的 React Native 工具模块了，但是大多数时候，你可能更需要的是自定义的原生 UI 组件。举例来说，我们需要一个跑马灯效果的文本组件，React Native 中并没有直接提供这样一个组件（当然你也可以使用 JavaScript 进行封装），我们可以直接封装一个这样效果的原生组件在 JavaScript 中调用。

12.5.1　封装 iOS 平台的原生 UI 组件

首先使用 Xcode 开发工具打开 iOS 工程，在我们前面创建的 RNTools 模块下新建一个类，如图 12-16 所示。

在弹出的设置窗口中将类命名为 FlashView，并使其继承自 RCTViewManager 类，如图 12-17 所示。

注意，不要忘记在 FlashView.h 文件中导入头文件，代码如下：

```
#import <React/RCTViewManager.h>
```

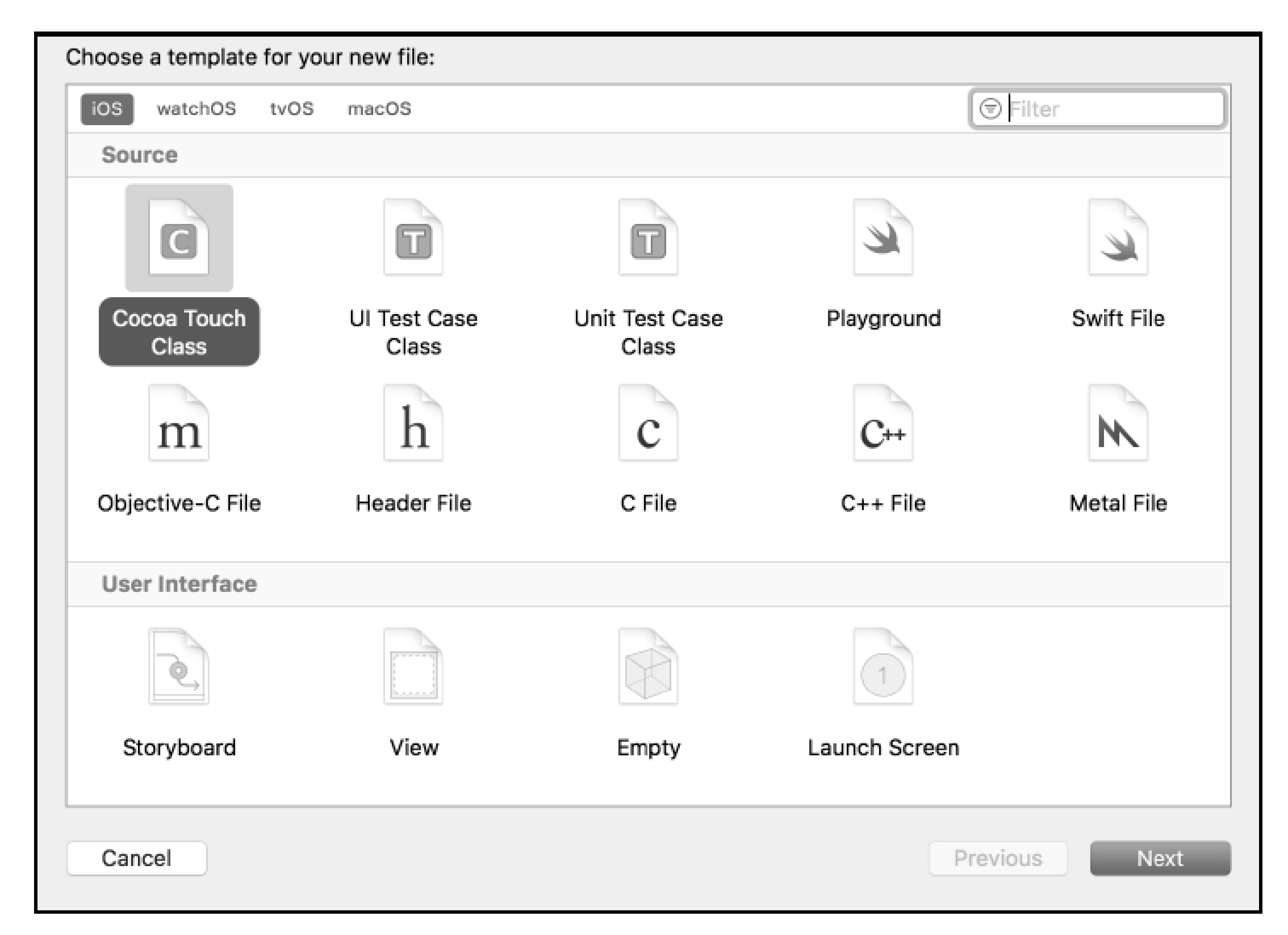


图 12-16 创建一个 Cocoa Touch 类文件



图 12-17 设置继承关系

RCTViewManager 类相当于一个视图控制器，其完成原生 UI 组件与 JavaScript 之间的桥接。下面我们还需要创建一个原生的组件来展现视图，使用上面的方法再次创建一个类文件，这次使其继承于原生的 UILabel 类，将其命名为 NativeFlashView。实现 FlashView.m 文件如下：

```
#import "FlashView.h"
#import "NativeFlashView.h"
@implementation FlashView
RCT_EXPORT_MODULE(FlashView)
-(UIView *)view{
   return [NativeFlashView new];
}
@end
```

创建原生 UI 组件至少需要两步：

步骤 01 将此控制器模块进行导出，即使用 RCT_EXPORT_MODULE 宏来完成。

步骤 02 实现 view 实例方法，这个方法将需要的原生视图返回。

下面我们可以在 JavaScript 工程中进行一个简单的测试，修改 ToolsManager.js 文件如下：

```
import { NativeModules,requireNativeComponent } from 'react-native';
export var ToolsManager = NativeModules.RNToolsManager;
export var FlashView = requireNativeComponent('FlashView',null);
```

requireNativeComponent 函数用来进行原生组件的引入。注意，这个函数接收两个参数，第 1 个参数为原生组件的名称，第 2 个参数为组件的描述对象，关于描述对象我们暂且搁置一边，后面会使用到。在 Demo 文件夹下新建一个命名为 NativeViewDemo.js 的文件，实现代码如下：

```
import React, { Component } from 'react';
import {View,Text} from 'react-native';
import {FlashView} from './ToolsManager';
export default class NativeViewDemo extends Component{
    render(){
        return(
            <FlashView style={{top:30,height:30,backgroundColor:'red'}}/>
        );
    }
}
```

修改 index.ios.js 文件后运行工程，可以看到自定义的原生组件已经显示在屏幕上，如图 12-18 所示。

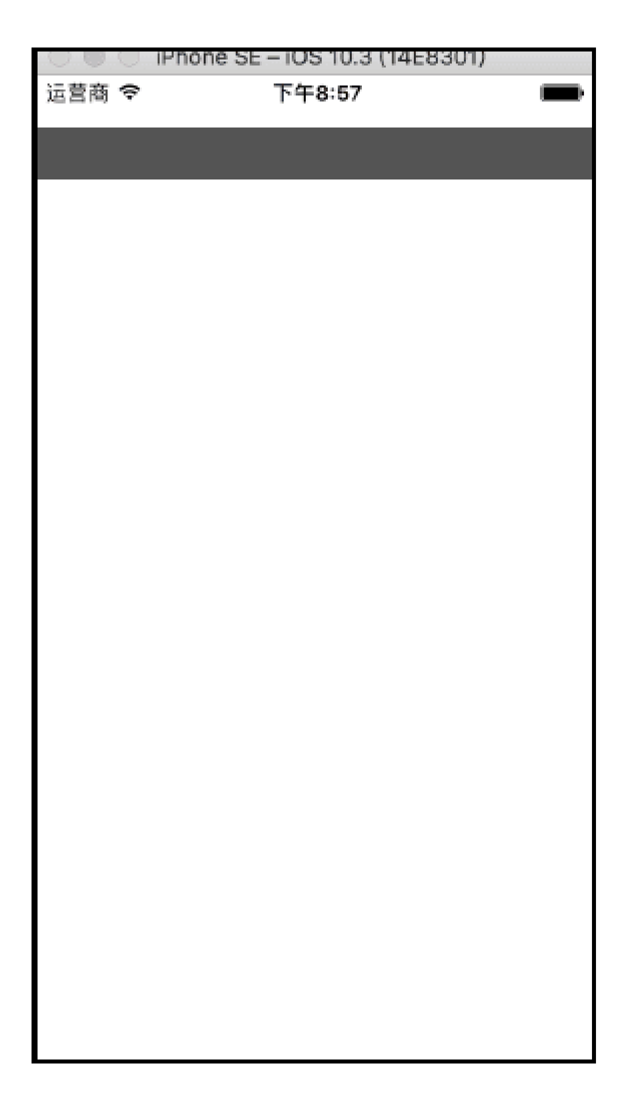


图 12-18　自定义原生组件

仅仅展现出来一个原生的视图远远不能满足我们的需求，我们需要能够设置组件上面的文字，这就需要使用到，将原生属性导出到 JavaScript 也十分简单，修改 FlashView.m 文件中的代码如下：

```
#import "FlashView.h"
#import "NativeFlashView.h"
@interface FlashView()

@property(nonatomic,strong)NSString * text;
@property(nonatomic,strong)NativeFlashView * nativeView;
@end
@implementation FlashView
RCT_EXPORT_MODULE(FlashView)
-(UIView *)view{
    return self.nativeView;
}
RCT_EXPORT_VIEW_PROPERTY(text, NSString*)
-(void)setText:(NSString *)text{
    self.nativeView.text = text;
}
-(NativeFlashView *)nativeView{
    if (!_nativeView) {
        _nativeView = [NativeFlashView new];
    }
    return _nativeView;
}
@end
```

看懂上面的代码需要一些原生开发经验，如果你做过 iOS 开发，那么上面的代码很容易理解，如果没有做过也没关系，核心在于 RCT_EXPORT_VIEW_PROPERTY 宏，它用来导出一个原生属性。

下面我们来修改一下 ToolsManager.js 文件中的代码，原则上讲，你已经可以直接在 JavaScript 代码中使用原生组件导出的属性，但是作为可复用组件的封装，不是所有使用者都会去原生代码中找所导出的属性有哪些，因此你可以将 JavaScript 组件再进行一次封装，将所导出的原生属性进行暴露，代码如下：

```
import React, { Component, PropTypes } from 'react';
import { NativeModules,requireNativeComponent } from 'react-native';
export var ToolsManager = NativeModules.RNToolsManager;
var FView = requireNativeComponent('FlashView',FlashView);
export class FlashView extends Component{
    static propTypes = {
        /*
         * 设置文本
        */
        text:PropTypes.string
    }
    render(){
        return(
```

```
                <FView {...this.props} />
            );
        }
    }
```

可以看到，requireNativeComponent 方法的第 2 个参数已经换成了 FlashView 对象，这样 React Native 就会对组件的属性及类型进行检查，便于开发调试与查错。修改 NativeViewDemo 类，代码如下：

```
    export default class NativeViewDemo extends Component{
        render(){
            return(
                <FlashView style={{top:30,height:30,backgroundColor:'red'}}
text="Hello World"/>
            );
        }
    }
```

再次编译运行工程，效果如图 12-19 所示。

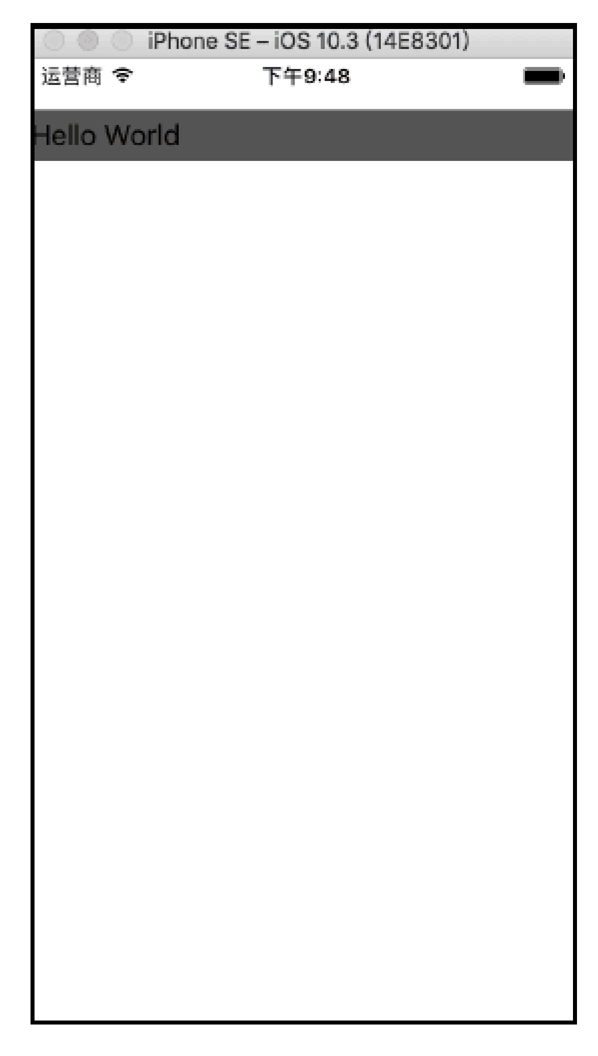


图 12-19　设置原生组件的属性

下面我们来实现跑马灯的需求，首先当用户单击文本组件时，跑马灯效果开启，再次单击则关闭跑马灯效果，因此需要为原生组件添加一个手势，先修改 NativeFlashView.h 文件，代码如下：

```
    #import <UIKit/UIKit.h>
    #import <React/UIView+React.h>
    @interface NativeFlashView : UILabel
    //开启
    -(void)start;
    //关闭
    -(void)stop;
    //JavaScript 回调
```

```
@property(nonatomic,copy)RCTBubblingEventBlock onPress;
@end
```

修改 NativeFlashView.m 文件，代码如下：

```
#import "NativeFlashView.h"
@interface NativeFlashView()
@property(nonatomic,strong)NSTimer * timer;
@end
@implementation NativeFlashView
- (instancetype)init
{
    self = [super init];
    if (self) {
        self.timer = [NSTimer scheduledTimerWithTimeInterval:1 target:self
selector:@selector(flash) userInfo:nil repeats:YES];
        self.timer.fireDate = [NSDate distantFuture];
    }
    return self;
}
-(void)flash{
    self.textColor = [UIColor colorWithRed:arc4random()%255/255.0
green:arc4random()%255/255.0 blue:arc4random()%255/255.0 alpha:1];
    self.backgroundColor = [UIColor colorWithRed:arc4random()%255/255.0
green:arc4random()%255/255.0 blue:arc4random()%255/255.0 alpha:1];
}
-(void)start{
    self.timer.fireDate = [NSDate distantPast];
}
-(void)stop{
    self.timer.fireDate = [NSDate distantFuture];
}
@end
```

修改 FlashView.m 文件，代码如下：

```
#import "FlashView.h"
#import "NativeFlashView.h"
@interface FlashView()
@property(nonatomic,strong)NSString * text;
@property(nonatomic,strong)NativeFlashView * nativeView;
//手势
@property(nonatomic,strong)UITapGestureRecognizer * tap;
//标记状态
@property(nonatomic,assign)BOOL flashing;
@end
@implementation FlashView
```

```
RCT_EXPORT_MODULE(FlashView)
-(UIView *)view{
    return self.nativeView;
}
RCT_EXPORT_VIEW_PROPERTY(text, NSString*)
RCT_EXPORT_VIEW_PROPERTY(onPress, RCTBubblingEventBlock)
-(void)setText:(NSString *)text{
    self.nativeView.text = text;
}
-(NativeFlashView *)nativeView{
    if (!_nativeView) {
        _nativeView = [NativeFlashView new];
        [_nativeView addGestureRecognizer:self.tap];
    }
    return _nativeView;
}
-(UITapGestureRecognizer *)tap{
    if (!_tap) {
        _tap = [[UITapGestureRecognizer alloc]initWithTarget:self
action:@selector(tapClick)];
    }
    return _tap;
}
-(void)tapClick{
    self.nativeView.onPress(@{@"flashing":@(self.flashing)});
}
//导出原生方法
RCT_REMAP_METHOD(start, start){
    [self.nativeView start];
    self.flashing = YES;
}
RCT_REMAP_METHOD(stop, stop){
    [self.nativeView stop];
    self.flashing = NO;
}
```

下面我们再来看一下 JavaScript 代码，首先修改 ToolsManager.js 文件，代码如下：

```
import React, { Component, PropTypes } from 'react';
import { NativeModules,requireNativeComponent } from 'react-native';
export var ToolsManager = NativeModules.RNToolsManager;
var FView = requireNativeComponent('FlashView',FlashView);
//导入方法模块
var FViewManager = NativeModules.FlashView;
export class FlashView extends Component{
    static propTypes = {
```

```
            /*
             * 设置文本
            */
            text:PropTypes.string,
            onPress:PropTypes.func
        }
        render(){
            return(
                <FView {...this.props} />
            );
        }
        start=()=>{
            FViewManager.start();
        }
        stop=()=>{
            FViewManager.stop();
        }
    }
```

修改 NativeViewDemo 类，代码如下：

```
    export default class NativeViewDemo extends Component{
        render(){
            return(
                <FlashView style={{top:30,height:30,backgroundColor:'red'}}
text="Hello World"
                onPress={(e)=>{
                    if (e.nativeEvent.flashing) {
                        this.refs.FlashView.stop();
                    }else{
                        this.refs.FlashView.start();
                    }
                }} ref="FlashView"/>
            );
        }
    }
```

再次编译运行工程，好好享受跑马灯的乐趣吧！

12.5.2 开发 Android 跑马灯组件

有了 iOS 平台的开发经验，再将其适配到 Android 平台就相对简单一些了。首先使用 Android Studio 工具打开 HelloWorld 工程，新建一个 Java 类，将其命名为 FlashView 并使其继承自 SimpleViewManager 类，编写代码如下：

```
    package com.helloworld;
    import android.graphics.Color;
```

```
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.widget.TextView;
import com.facebook.react.bridge.Arguments;
import com.facebook.react.bridge.ReactMethod;
import com.facebook.react.bridge.WritableMap;
import com.facebook.react.uimanager.SimpleViewManager;
import com.facebook.react.uimanager.ThemedReactContext;
import com.facebook.react.uimanager.annotations.ReactProp;
import com.facebook.react.uimanager.events.RCTEventEmitter;
import java.util.Timer;
import java.util.TimerTask;
//需要注意，父类后的中括号中需要写上要包装的组件类
public class FlashView extends SimpleViewManager<TextView> {
    //上下文
    private static ThemedReactContext context;
    //原生 UI 组件，这里使用原生的 TextView，当然也可以是自定义的
    private static TextView nativeView;
    //跑马灯开关
    static public boolean isflashing = false;
    //定时器
    private Timer timer;
    //进行跑马灯开关操作的方法
    Handler handler = new Handler(Looper.getMainLooper(),new Handler.Callback() {
        @Override
        public boolean handleMessage(Message msg) {
            if (msg.what == 1) {
                if (isflashing) {
                    nativeView.setTextColor(Color.rgb((int) (Math.random() *
255), (int) (Math.random() * 255), (int) (Math.random() * 255)));
                    nativeView.setBackgroundColor(Color.rgb((int)
(Math.random() * 255), (int) (Math.random() * 255), (int) (Math.random() * 255)));
                    Log.e("e","1");
                }
            }else if(msg.what==2){
                if (!isflashing) {
                    if (timer==null){
                        timer = new Timer();
                        timer.schedule(task, 0, 1000);
                    }
                }
```

```
                    isflashing = true;
                }else{
                    isflashing = false;
                }
                return true;
            }
        });
        //定时器任务对象
        TimerTask task = new TimerTask() {
            @Override
            public void run() {
                // 需要做的事:发送消息
                Message msg = new Message();
                msg.what = 1;
                handler.sendMessage(msg);
            }
        };
        //实现这个方法来进行导出原生组件名称的设置
        @Override
        public String getName() {
            return "FlashView";
        }
        //实现这个方法来进行原生组件的创建
        @Override
        protected TextView createViewInstance(final ThemedReactContext
reactContext) {
            this.context = reactContext;
            this.nativeView = new TextView(context);
            final FlashView self = this;
            //设置手势监听
            this.nativeView.setOnTouchListener(new View.OnTouchListener() {
                @Override
                public boolean onTouch(View v, MotionEvent event) {
                    if(event.getAction() != MotionEvent.ACTION_UP){
                        return true;
                    }
                    WritableMap param = Arguments.createMap();
                    param.putBoolean("flashing",isflashing);
                    //调用 JavaScript 端的函数属性
                    self.context.getJSModule(RCTEventEmitter.class).receiveEvent(
                            nativeView.getId(),
                            "topChange",
                            param
                    );
                    return true;
```

```
            }
        });
        return this.nativeView;
    }
    //导出属性
    @ReactProp(name="text")
    public void setText(TextView label,String text){
        this.nativeView.setText(text);
    }

    //导出方法
    @ReactMethod
    public void start(){
      Message m = new Message();
        m.what = 2;
        handler.sendMessage(m);
    }
    @ReactMethod
    public void stop(){
        Message m = new Message();
        m.what = 3;
        handler.sendMessage(m);
    }
}
```

看懂上面的代码需要你有一定原生开发的能力，虽然上面的代码看似复杂，但是其实核心部分很清晰。注意，self.context.getJSModule(RCTEventEmitter.class).receiveEvent(nativeView.getId(), "topChange", param);这部分代码是对 JavaScript 组件属性中的函数进行调用，与 iOS 略有不同，getId()函数用来关联原生与 JavaScriptUI 组件，topChange 为导出方法名称，但是需要特别注意，这个名称在 JavaScript 端会被映射成 onChange 属性，param 为我们要传递的原生参数。就如在 Android 平台开发原生模块一样，你需要再创建一个包装类，新建一个 Java 类文件，将其命名为 FlashViewPackage 并使其继承自 ReactPackage 类，实现代码如下：

```
package com.helloworld;
import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class FlashViewPackage  implements ReactPackage{
    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext
reactContext) {
```

```
            List<NativeModule> list = new ArrayList<>();
            list.add(new FlashView());
            return list;
        }
        @Override
        public List<ViewManager> createViewManagers(ReactApplicationContext
reactContext) {
            return Arrays.<ViewManager>asList(
                    new FlashView()
            );
        }
    }
```

最后不要忘了，在 MainApplication 类中将 FlashViewPackage 进行注册，代码如下：

```
        @Override
        protected List<ReactPackage> getPackages() {
          return Arrays.<ReactPackage>asList(
                  new MainReactPackage(), new RNToolsManagerPackage(),
                  new FlashViewPackage()
          );
        }
```

最后，需要将 JavaScript 工程中的 ToolsManager 类进行一点点修改，我们刚才提到，Android 中调用的 topChange 方法会映射成 JavaScript 中的 onChange 属性，我们需要对其进行一下兼容：

```
    export class FlashView extends Component{
        static propTypes = {
            /*
             * 设置文本
            */
            text:PropTypes.string,
            onPress:PropTypes.func
        }
        render(){
            return(
                <FView {...this.props} onChange={this.props.onPress}/>
            );
        }
        start=()=>{
            FViewManager.start();
        }
        stop=()=>{
            FViewManager.stop();
        }
    }
```

现在，在 Android 平台编译并运营你的工程，你开发的第一个可复用的原生跑马灯组件就完成了。

12.6　在原生工程中嵌入 React Native 模块

本章中的许多内容都属于 React Native 技术的高级内容，如果你无法完全掌握它们也不要心急，很多知识是在你拥有了足够的开发经验后融会贯通的。本书前面介绍的所有内容都是基于完整的 React Native 工程的，有些时候，你可能已经有了一个成型的原生项目，你想将其中的某一块使用 React Native 进行实现，以达到复用和快速迭代的目的，这对 React Native 来说也十分容易。

12.6.1　将 iOS 工程的某个模块进行 React Native 化

首先，你需要有一个 React Native 原生工程，使用 Xcode 开发工具创建一个新的工程，如图 12-20 所示。

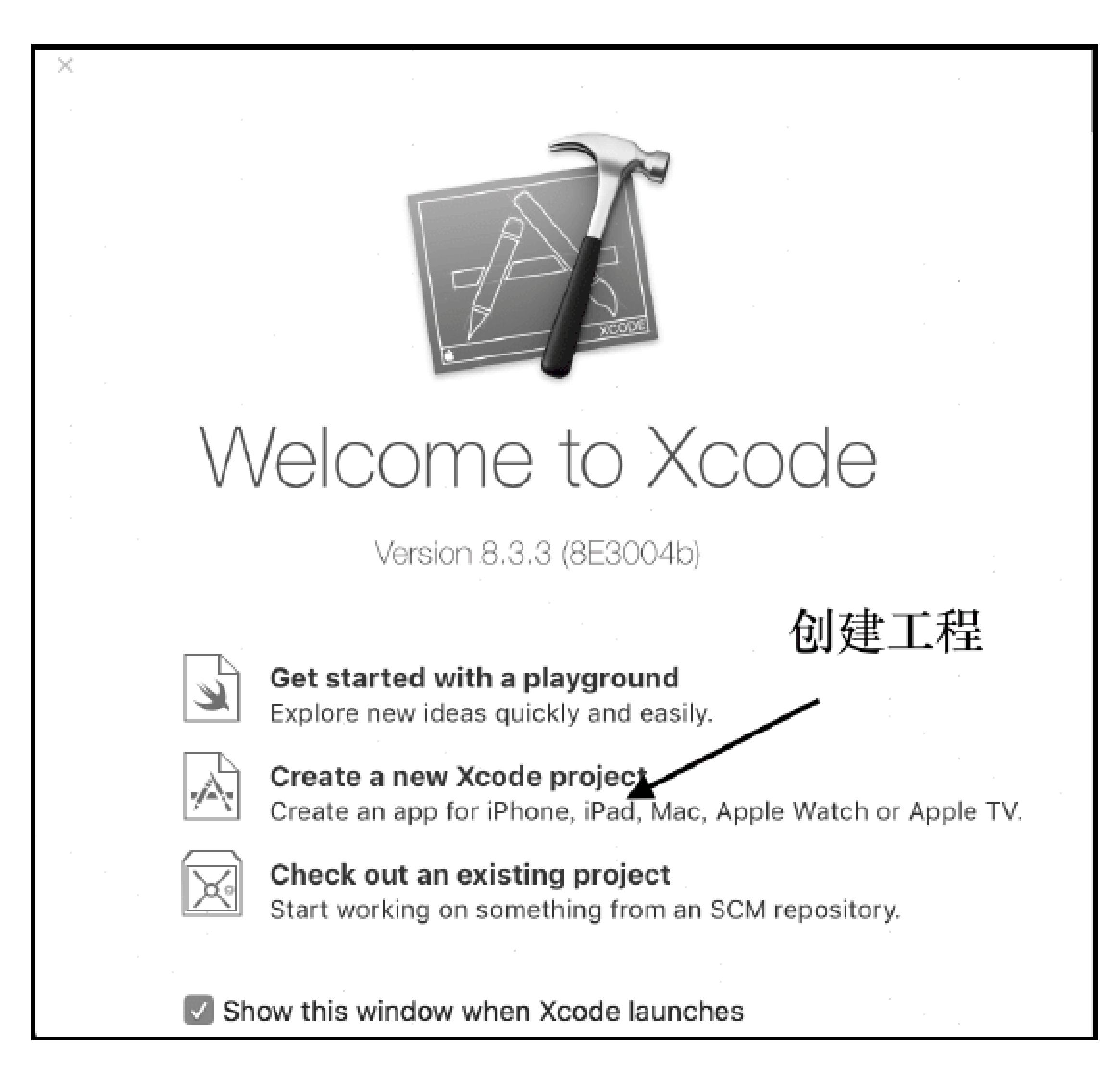


图 12-20　创建新工程

将创建的工程命名为 React NativeModule，新创建的工程可以直接运行，单击 Xcode 开发工具左上角的运行按钮即可在模拟器上运行工程。我们刚才所建的工程是一个空的项目，如果运行，你会看到设备界面上一片空白，下面我们来向界面中添加一个按钮，在 ViewController 类的 viewDidLoad 方法中编写如下代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIButton * button = [UIButton buttonWithType:UIButtonTypeSystem];
    [button setTitle:@"Hello RN" forState:UIControlStateNormal];
    button.frame = CGRectMake(self.view.frame.size.width/2-50, 200, 100, 30);
    [self.view addSubview:button];
```

```
        [button addTarget:self action:@selector(gotoRN) forControlEvents:
UIControlEventTouchUpInside];
    }
```

上面的代码创建了一个按钮，并为此按钮添加了一个名为 gotoRN 的单击事件，我们可以暂时不实现这个方法，运行工程，效果如图 12-21 所示。

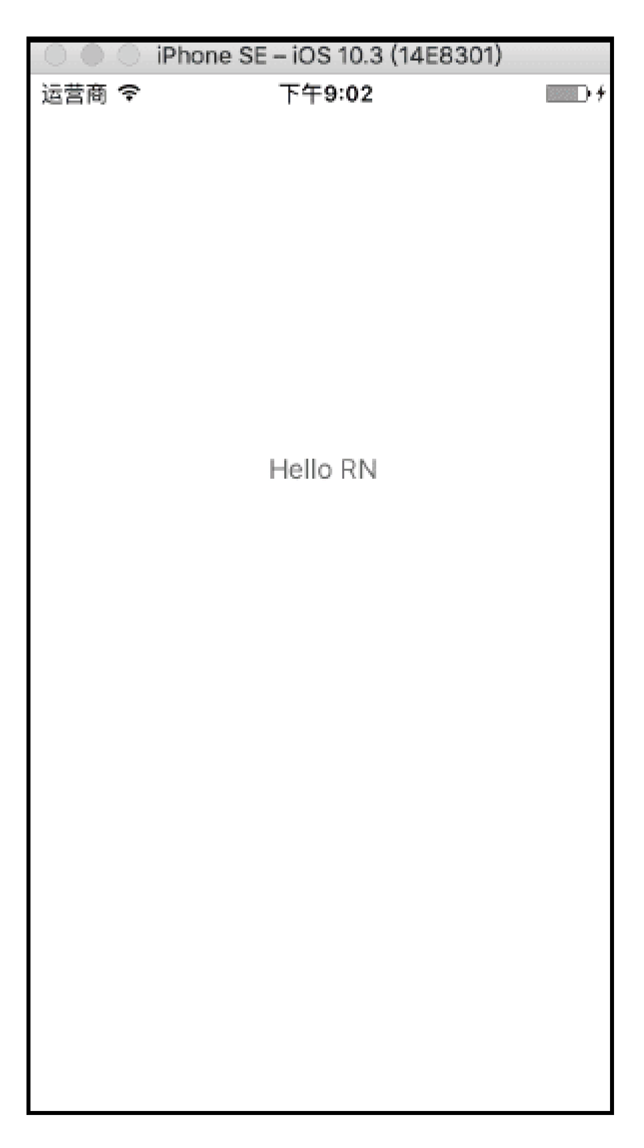


图 12-21　原生工程界面

下面我们实现当单击 Hello RN 按钮后跳转到一个 React Native 界面。首先需要构建一个 React Native 工程，在桌面上新建一个目录，命名为 RNModule，在其中再新建一个文件夹，命名为 ios，接着将原生 iOS 工程中的所有文件移动到这个 ios 文件夹下，然后在 RNModule 文件夹下新建一个命名为 package.json 的文件，编写如下代码：

```
    {
      "name": "React NativeModule",
      "version": "0.0.1",
      "private": true,
      "scripts": {
        "start": "node node_modules/react-native/local-cli/cli.js start"
      },
      "dependencies": {
        "react": "16.0.0-alpha.6",
        "react-native": "0.44.3"
      }
    }
```

注意，示例中使用的 React Native 版本为 0.44.3，某些 React Native 版本可能并不支持嵌入原生模块中。之后使用终端在 RNModule 文件夹下执行 yarn install 命令，完成之后，RNModule 文件夹下会多出一个 node_modules 目录，这个目录中存放着所有需要使用到的框架。

完成了上面的过程，我们需要对原生的 iOS 工程进行配置，首先需要安装 CocoaPods 管理工具，这个工具用来管理 iOS 工程的第三方类库，类似于 Node 中的 npm 工具。关于 CocoaPods 的安装，这里就不再赘述了，互联网上有很多的资料可以参考。使用终端在 ios 目录下执行如下命令进行 CocoaPods 工程的初始化：

```
pod init
```

执行完成 pod init 命令后，ios 文件夹中会多出一个 Podfile 文件，在其中编写如下代码：

```
target 'React NativeModule' do
  pod 'React', :path => '../node_modules/react-native', :subspecs => [
    'Core',
    'DevSupport',
    'RCTText',
    'RCTNetwork',
    'RCTWebSocket',
    # 在这里继续添加你所需要的模块
  ]
  pod "Yoga", :path => "../node_modules/react-native/ReactCommon/yoga"
end
```

上面我们引入 React Native 中的核心模块，网络模块、开发者调试模块和文本组件 Text 模块，如果你需要引入其他的 React Native 模块，可以继续导入。关于 React Native 所有模块的名称定义在 node_modules→react-native→React.podspec 文件中。之后使用终端在 ios 目录下执行如下命令：

```
pod install
```

完成上面的命令后，你的工程结构应该如图 12-22 所示。

需要注意，执行完 pod install 命令后，你需要使用 React NativeModule.xcworkspace 文件打开原生工程。

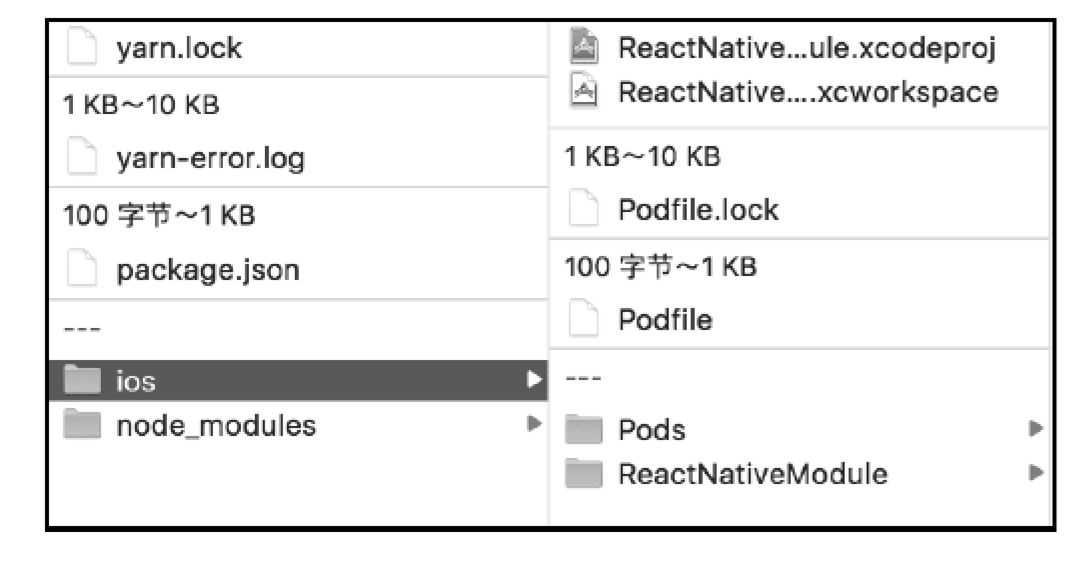


图 12-22　工程目录结构

下面我们来编写一个简单的 React Native 界面，在 RNModule 目录下新建一个命名为 index.ios.js 的文件，在其中编写如下代码：

```
import React from 'react';
import {
  AppRegistry,
  StyleSheet,
  Text,
  View
} from 'react-native';
class RNView extends React.Component {
  render() {
    return (
```

```
        <View style={styles.container}>
          <Text style={styles.text}>
              Welcome to React Native!
          </Text>
        </View>
      );
    }
  }
  const styles = StyleSheet.create({
    container: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center',
      backgroundColor: '#FFFFFF',
    },
    text: {
      fontSize: 20,
      textAlign: 'center',
      margin: 10,
    }
  });
  // 整体 js 模块的名称
  AppRegistry.registerComponent('React NativeModule', () => RNView);
```

上面的代码创建了一个简单的文本，并将整个模块进行注册。在 iOS 原生工程的 ViewController.m 文件中导入如下头文件：

```
#import <React/RCTRootView.h>
```

实现我们前面空下的 gotoRN 方法：

```
-(void)gotoRN{
    //创建 React Native 模块本地地址
    NSURL *jsCodeLocation = [NSURL

URLWithString:@"http://localhost:8081/index.ios.bundle?platform=ios"];
    //创建 React Native 视图
    RCTRootView *rootView =
    [[RCTRootView alloc] initWithBundleURL : jsCodeLocation
                     moduleName       : @"React NativeModule"
                     initialProperties : nil
                     launchOptions     : nil];
    UIViewController *vc = [[UIViewController alloc] init];
    vc.view = rootView;
    [self presentViewController:vc animated:YES completion:nil];
}
```

要运行工程，首先使用终端在 RNModule 目录下执行如下命令来开启本地服务：

```
npm start
```

之后在 RNMudule 目录下执行下面的命令来运行 iOS 原生工程：

```
react-native run-ios
```

单击界面上的 Hello RN 按钮，可以看到界面跳转到了 React Native 模块，如图 12-23 所示。



图 12-23　React Native 模块界面

12.6.2　将 Android 工程的某个模块进行 React Native 化

我们仿照 iOS 工程的模式来进行 Android 工程原生模块的 React Native 化，首先使用 Android Studio 工具创建一个空的 Android 工程，在选择工程模板时，选择 Empty Activity，如图 12-24 所示。

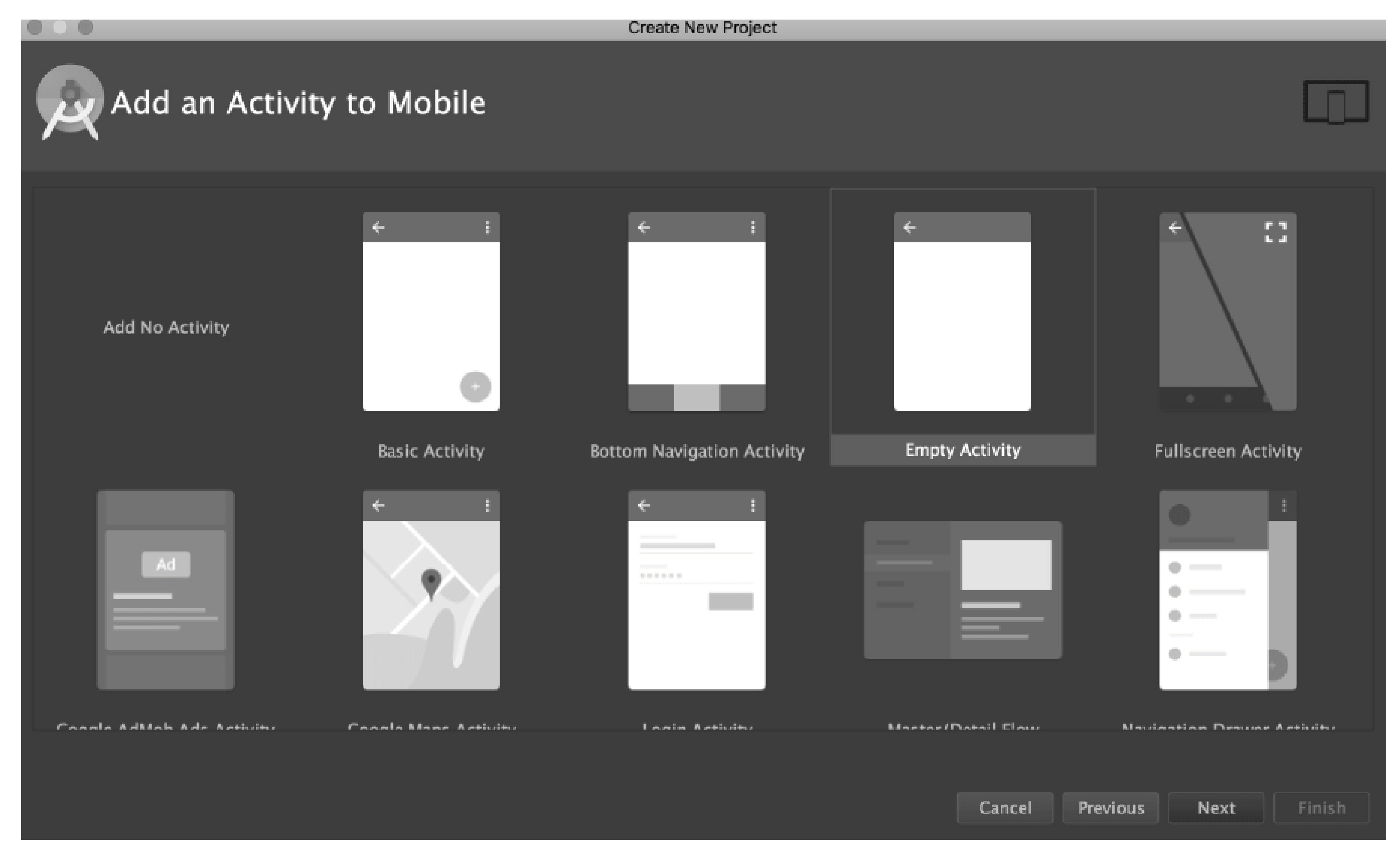


图 12-24　创建空的 Android 工程

创建完工程后，无须编写任何代码，在模拟器上进行项目的运行，效果如图 12-25 所示。

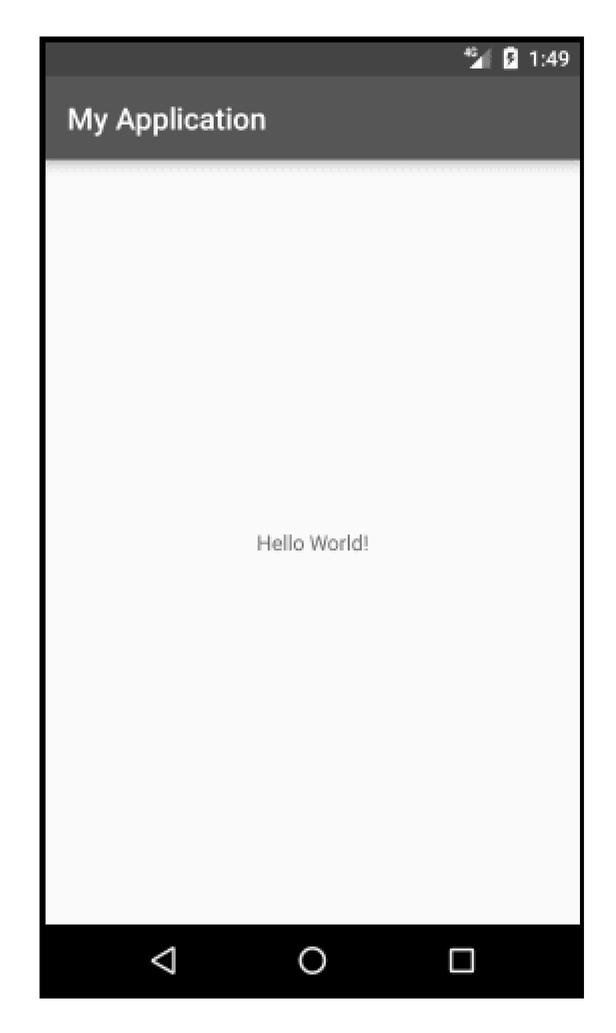


图 12-25　HelloWorld 工程界面

JavaScript 工程先不需要做任何修改，在根目录下新建一个命名为 index.android.js 的文件，并将 index.ios.js 文件中的所有内容复制进来即可。下面进行 Android 原生工程的配置，首先在 Module 下的 build.gradle 文件中添加依赖：

```
    apply plugin: 'com.android.application'
    android {
        compileSdkVersion 23
        buildToolsVersion "25.0.0"
        defaultConfig {
            applicationId "com.example.jaki.myapplication"
            minSdkVersion 16
            targetSdkVersion 23
            versionCode 1
            versionName "1.0"
            testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
            ndk {
                abiFilters "armeabi-v7a", "x86"
            }
            packagingOptions {
                exclude "lib/arm64-v8a/librealm-jni.so"
            }
        }
        buildTypes {
            release {
                minifyEnabled false
                proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
```

```
            }
        }
        configurations.all {
            resolutionStrategy.force 'com.google.code.findbugs:jsr305:1.3.9'
        }
    }
    dependencies {
        compile fileTree(dir: 'libs', include: ['*.jar'])
        androidTestCompile('com.android.support.test.espresso:
espresso-core:2.2.2', {
            exclude group: 'com.android.support', module: 'support-annotations'
        })
        compile 'com.android.support:appcompat-v7:23.0.1'
        compile 'com.android.support.constraint:constraint-layout:1.0.2'
        testCompile 'junit:junit:4.12'
        compile "com.facebook.react:react-native:0.44.3"
    }
```

修改 Product 下的 build.gradle 文件，代码如下：

```
    buildscript {
        repositories {
            jcenter()
        }
        dependencies {
            classpath 'com.android.tools.build:gradle:2.3.0'
            // NOTE: Do not place your application dependencies here; they belong
            // in the individual module build.gradle files
        }
    }
    allprojects {
        repositories {
            jcenter()
        }
    }
    allprojects {
        repositories {
            maven {
                // All of React Native (JS, Android binaries) is installed from npm
                url "$rootDir/../node_modules/react-native/android"
            }
        }
    }
    task clean(type: Delete) {
        delete rootProject.buildDir
    }
```

在工程中新建一个命名为 RNActivity 的 Java 类，使其继承自 Activity，在其中编写如下代码：

```
    package com.example.jaki.myapplication;
    import android.app.Activity;
    import android.os.Bundle;
    import android.view.KeyEvent;
    import com.facebook.react.ReactInstanceManager;
    import com.facebook.react.ReactRootView;
    import com.facebook.react.common.LifecycleState;
    import com.facebook.react.modules.core.DefaultHardwareBackBtnHandler;
    import com.facebook.react.shell.MainReactPackage;
    public class RNActivity extends Activity implements
DefaultHardwareBackBtnHandler {
        private ReactRootView mReactRootView;
        private ReactInstanceManager mReactInstanceManager;
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            mReactRootView = new ReactRootView(this);
            mReactInstanceManager = ReactInstanceManager.builder()
                    .setApplication(getApplication())
                    .setBundleAssetName("index.android.bundle")
                    .setJSMainModuleName("index.android")
                    .addPackage(new MainReactPackage())
                    .setUseDeveloperSupport(BuildConfig.DEBUG)
                    .setInitialLifecycleState(LifecycleState.RESUMED)
                    .build();
            // 注意这里的 MyReact NativeApp 必须对应“index.android.js”中
            // “AppRegistry.registerComponent()”的第一个参数
            mReactRootView.startReactApplication(mReactInstanceManager, "React
NativeModule", null);
            setContentView(mReactRootView);
        }
        @Override
        public void invokeDefaultOnBackPressed() {
            super.onBackPressed();
        }
        @Override
        protected void onPause() {
            super.onPause();
            if (mReactInstanceManager != null) {
                mReactInstanceManager.onHostPause(this);
            }
        }
        @Override
```

```
    protected void onResume() {
        super.onResume();
        if (mReactInstanceManager != null) {
            mReactInstanceManager.onHostResume(this, this);
        }
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mReactInstanceManager != null) {
            mReactInstanceManager.onHostDestroy();
        }
    }
    @Override
    public void onBackPressed() {
        if (mReactInstanceManager != null) {
            mReactInstanceManager.onBackPressed();
        } else {
            super.onBackPressed();
        }
    }
    @Override
    public boolean onKeyUp(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_MENU && mReactInstanceManager != null) {
            mReactInstanceManager.showDevOptionsDialog();
            return true;
        }
        return super.onKeyUp(keyCode, event);
    }
}
```

修改 MainActivity 中的代码：

```
package com.example.jaki.myapplication;
import android.content.Intent;
import android.os.Build;
import android.provider.Settings;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity  {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
            setContentView(R.layout.activity_main);
            TextView text = (TextView) findViewById(R.id.textView);
            final MainActivity self = this;
            text.setOnTouchListener(new View.OnTouchListener() {
                @Override
                public boolean onTouch(View v, MotionEvent event) {
                    if (event.getAction() == MotionEvent.ACTION_UP){
                        if (Build.VERSION.SDK_INT >= 23) {
                            if(!Settings.canDrawOverlays(self)) {
                                Intent intent = new
Intent(Settings.ACTION_MANAGE_OVERLAY_PERMISSION);
                                startActivity(intent);
                            } else {
                                //绘 ui 代码, 这里说明 6.0 系统已经有权限了
                                Intent intent = new Intent(MainActivity.this,
RNActivity.class);
                                startActivity(intent);
                            }
                        } else {
                            //绘 ui 代码,这里 Android 6.0 以下的系统直接绘出即可
                            Intent intent = new Intent(MainActivity.this,
RNActivity.class);
                            startActivity(intent);
                        }
                    }
                    return  true;
                }
            });
        }
    }
```

最后，需要在 AndroidManifest.xml 文件中进行权限的申请和 Activity 的注册，代码如下：

```
    <?xml version="1.0" encoding="utf-8"?>
    <manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.example.jaki.myapplication">
        <uses-permission android:name="android.permission.INTERNET" />
        <uses-permission android:name=
"android.permission.SYSTEM_OVERLAY_WINDOW"/>
        <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
        <application
            android:allowBackup="true"
            android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"
            android:supportsRtl="true"
            android:theme="@style/AppTheme">
```

```
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=
"com.facebook.react.devsupport.DevSettingsActivity" />
        <activity
            android:name=".RNActivity"
            android:label="@string/app_name"
            android:theme="@style/Theme.AppCompat.Light.NoActionBar">
        </activity>
    </application>
</manifest>
```

开启 React Native 的 Node 服务，并编译运行工程，你会看到在 Android 平台上的效果与 iOS 平台一致。

12.7　在真机上运行 React Native 工程

前面我们一直在使用模拟器来进行 React Native 项目的开发其实也可以直接在 iPhone 手机或 Android 手机上进行项目的运行。在 iOS 真机上运行工程十分容易，将手机用数据线连接电脑，使用 Xcode 开发工具打开工程。若要在 iOS 真机上进行项目的运行，首先需要有一个 AppID 账号，并在 Xcode 工具栏中的 Perfrences→accounts 中进行登录，之后用工程的运行设备选择你的手机，单击运行按钮即可，如图 12-26 所示。



图 12-26　在真机设备上运行 React Native 工程

需要注意，你的手机所在的网络必须与电脑在同一网段中（例如连同一个 WIFI），如果想在真机中唤出开发者菜单，可以摇晃手机。

在 Android 设备上运行工程只需要通过数据线将手机与电脑连接，并且保证只有一个设备连接，关闭所有模拟器，使用下面的命令进行安装即可：

```
react-native run-android
```